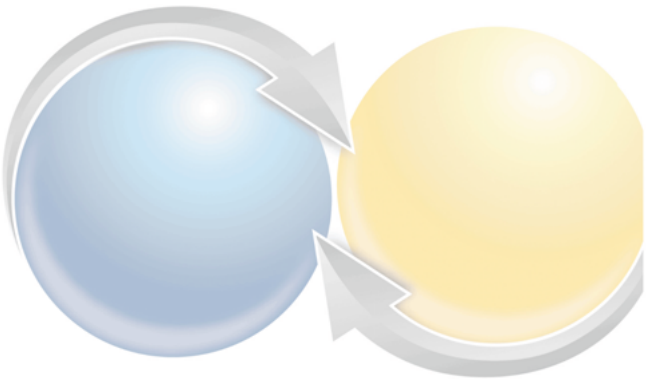
Livelink WCM Server

WCM Java API Programmer's Manual

This manual describes the Java programming interface, which allows the functionality of the WCM servers to be used via external programs. Highlighted topics include:

- implementation basics
- access to users and objects from the WCM system and from Livelink
- event processing in the WCM system
- access to system information and administration via the WCM Java API
- application examples of using the WCM Java API



OPEN TEXT CORPORATION

Great Minds Working Together™

Copyright 2006 by Open Text Corporation.

The copyright to these materials and any accompanying software is owned, without reservation, by Open Text Corporation. These materials and any accompanying software may not be copied in whole or part without the express, written permission of Open Text Corporation.

Open Text Corporation is the owner of the trademarks Open Text, 'Great Minds Working Together', Livelink, and MeetingZone among others. This list is not exhaustive. All other products or company names are used for identification purposes only, and are trademarks of their respective owners. All rights reserved.

Open Text Corporation provides certain warranties and limitations in connection with the software that this document describes. For information about these warranties and limitations, refer to the license agreement entered into between the licensee and Open Text Corporation.

Contacting Us

Open Text Corporation
Corporate Headquarters
275 Frank Tompa Drive,
Waterloo, Ontario, Canada
N2L 0A1

(519) 888-7111

If you subscribe to our Customer Assistance Program or would like more information about the support program, write to Open Text Corporation's Customer Support at support@opentext.com or telephone (800) 540-7292 or (519) 888-9933. Our support hours are Monday through Friday, 8:30 a.m. to 8 p.m. (EST).

If you have comments or suggestions regarding this documentation, write to the Open Text Corporation Publications Group at documentation@opentext.com.

For more information about Open Text Corporation's products and services, visit our home page at http://www.opentext.com.

© 2006 IXOS SOFTWARE AG

Werner-v.-Siemens-Ring 20
85630 Grasbrunn, Germany

Tel.: +49 (89) 4629-0
Fax: +49 (89) 4629-1199
eMail: office@ixos.de
Internet: http://www.ixos.com

All rights reserved, including those regarding reproduction, copying or other use or communication of the contents of this document or parts thereof. No part of this publication may be reproduced, transmitted to third parties, processed using electronic retrieval systems, copied, distributed or used for public demonstration in any form without the written consent of IXOS SOFTWARE AG. We reserve the right to update or modify the contents. Any and all information that appears within illustrations of screenshots is

provided coincidentally to better demonstrate the functioning of the software. IXOS SOFTWARE AG hereby declares that this information reflects no statistics of nor has any validity for any existing company. This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org/) and software developed by the Apache Software Foundation (http://www.apache.org/).

Trademarks IXOS: IXOS SOFTWARE AG.

SAP®, R/3® and SAP ArchiveLink® are registered trademarks of SAP AG.

Microsoft®, Microsoft Windows NT® and the names of further Microsoft products are registered trademarks of Microsoft Corporation.

Acrobat Reader Copyright © 1987 Adobe Systems Incorporated. All rights reserved. Adobe and Acrobat are trademarks of Adobe Systems Incorporated which may be registered in certain jurisdictions.

Siebel® is a registered trademark by Siebel Systems, Inc.

Other product names are used only to identify the products and they may be registered trademarks of the relevant manufacturers.

Copyright © 2006 Gauss Interprise AG Hamburg, Gauss Interprise, Inc. Irvine, California. All rights reserved worldwide.

This document and the related software are property of Gauss Interprise AG or its suppliers and protected by copyright and other laws. They are distributed under licenses restricting their use, copying, distribution, and decompilation. Neither receipt nor possession of this document confers or transfers any right to reproduce or disclose any part of the contents hereof. No part of this document may be reproduced in any form by any means without prior written authorization of Gauss Interprise AG or Gauss Interprise, Inc.

Moreover, the regulations of the software license agreement apply to this documentation.

All brand names and trademarks mentioned are the property of their respective owners.

http://www.opentext.com/bridging/gauss.html

Program version: Livelink Web Content Management Server™ (Content Server) 9.5.1

Document version: En-01

Publication date: May 2006

Table of Contents

List of Figures **7**

List of Tables **8**

Chapter 1 Introduction **11**

About this Manual 12

Typographic Conventions 16

Chapter 2 Implementation Basics **19**

Architecture of the WCM Java API 20

Server Agents 22

Remote API 30

Authentication and Contexts 36

Error Handling 40

Localized Messages 42

Pool Management 44

Chapter 3 User Administration **51**

The `Principal` Interface 53

Editing Principals 54

Setting WCM Attributes in LDAP 61

Editing Rights of Principals 64

Editing Functional Areas 71

Editing Principal Assignments 74

Saving User Settings 76

Chapter 4 Event Processing 81
Normal Events 83
Preparation Events 89
Event Listeners 90
Event Dispatchers 92
Chapter 5 Object Management 95
Conditions for Editing Objects 96
Actions of the `ObjectHandler` Interface 108
The `ObjectHandlerUtil` Utility Class 120
Editing Object Data – The `ObjectData` Interface 121
Editing Special Attributes 128
Searching WCM Objects 133
Editing Access Control Lists 141
Performing Workflow Actions 142
Converting Objects 146
E-Mail Support 149
Chapter 6 System Information and Administration 157
Retrieving Information on the WCM System 160
Information on the Deployment 164
Retrieving and Setting Run Levels 173
Logs, Reports, and Tracing 174
Chapter 7 Accessing Livelink Systems 181
Livelink User Administration 183
Livelink Object Management 184
Relators 196

Chapter 8 Application Examples 205

Basic Class: `ExampleAgent` 207

Automatic Submission 209

Vetoing Metadata Changes 212

Monitoring Deployment Events 215

Converting Special Characters 221

Multi-Language Websites 228

Glossary 237

Index 243

List of Figures

Fig. 1 –	Basic WCM Java API interfaces	20
Fig. 2 –	Implementation of a server agent	22
Fig. 3 –	Managing server agents in the Admin client	23
Fig. 4 –	Configuring the demo agent with nodes	25
Fig. 5 –	Configuring the demo agent without nodes	26
Fig. 6 –	Implementation on the basis of the remote API	31
Fig. 7 –	LDAP schema – example	63
Fig. 8 –	Object management components	95
Fig. 9 –	Staging for processing a WCM object	101
Fig. 10 –	Class diagram – `Key` and `Value` interfaces	131
Fig. 11 –	Class diagram – Filter classes and interfaces	134
Fig. 12 –	Class diagram – Livelink principals	183
Fig. 13 –	Class diagram – Livelink object types	190

List of Tables

Table 1 –	Available functions of the Livelink WCM Server APIs	14
Table 2 –	Criteria for searching principals and examples	58
Table 3 –	The individual administration rights	66
Table 4 –	Constants of the access rights for WCM objects	69
Table 5 –	Object state constants	97
Table 6 –	Object statuses and possible actions	98
Table 7 –	Visible statuses of WCM objects in the data storage views	102
Table 8 –	Workflow transitions and required staging actions	104
Table 9 –	Main actions of the `ObjectHandler` interface	109
Table 10 –	Synchronous and Asynchronous Methods	111
Table 11 –	Attribute types and filters	135
Table 12 –	Activity types	143
Table 13 –	Constants for tracing settings (`TraceFilter` interface)	177

CHAPTER 1
Introduction

The WCM Java API provides access to the functions of Livelink Web Content Management Server™ (Livelink WCM Server for short) via a Java-based programming interface. Using the WCM Java API classes and interfaces, you can access the WCM-managed content and use it in the context of your own software development. Possible applications include integrating third-party systems or monitoring actions with the option of preventing an action from being performed (veto mechanism).

The central functions of Livelink WCM Server that you may access via the WCM Java API include:

- object management, e.g. adding, editing, and deleting WCM objects
- user management, e.g. adding, editing, and deleting users
- system administration, e.g. retrieving information on the websites, servers, and deployment systems of the WCM system, including influencing the system status

Access to the functions of the WCM Java API is provided by server agents which run in a WCM server. Alternately, you can use the remote API which enables Java programs running on any computer to access the WCM Java API.

Important! Incorrect use of the programming interface described in this manual may lead to errors in the WCM system including system crashes and data loss. Incorrect programming can also cause problems concerning performance and system resources. For this reason it is essential to test the developed software with regard to correctness, stability, robustness, and performance before putting it to productive operation.

Gauss Interprise AG cannot assume any liability for the correct functionality of the developed software. Our Professional Services Group can help you plan and implement solutions. This may help you avoid problems right from the start.

About this Manual

This documentation is designed for software developers with the necessary knowledge of the programming language Java and the functionality of Livelink WCM Server.

Note: The individual classes, interfaces, and methods of the WCM Java API are only briefly described here. Please refer to the online documentation (Javadoc) for complete and detailed descriptions. The Javadoc documentation is located in the directory **{WCM installation directory}\documentation\javadoc**.

The contents of this manual are organized as follows:

- Chapter 2 “Implementation Basics” describes the basic procedure for programming server agents via the WCM Java API functionalities. This includes, for example, topics such as starting and stopping agents, authentication, context management, and error handling.

- Chapter 3 “User Administration” describes the user management functions available to you via the WCM Java API. This includes topics such as adding, deleting, and editing users, groups, and roles as well as their rights and assignments.
- Chapter 4 “Event Processing” deals with the basics of event processing in the WCM system.
- Chapter 5 “Object Management” introduces the basics of editing WCM objects.
- Chapter 6 “System Information and Administration” describes how to use the WCM Java API functions to retrieve information about the components and the status of the WCM system and to change the system status.
- Chapter 7 “Accessing Livelink Systems” describes how to access Livelink systems and their objects via the WCM Java API.
- Chapter 8 “Application Examples” contains some examples of using the WCM Java API.

The programming interface described here is a component of Livelink WCM Server. In addition to this Programmer’s Manual, the following sources provide information:

- Content Client User Manual: This manual provides detailed instructions on all tasks involved in editorial work on websites.
- Livelink WCM Server Administrator Manual: This document explains how to configure and administer WCM systems. Moreover, it contains a detailed description of the technical concepts of Livelink WCM Server.

Livelink WCM Server offers flexible programming interfaces for using the core functionality of Livelink WCM Server via external programs. The following table provides an overview of the different interfaces and the available functions.

Table 1 – Available functions of the Livelink WCM Server APIs

Functionality of the	WCM Java API	Remote API	Portal Manager API	WCM WebServices
AdminHandler	✓	✓	✓ read-only by means of VipUserBean	
AttributeHandler	✓	✓	✓ by means of VipWebsiteBean	✓ by means of VipWebService_Port
EventDispatcher	✓	✓		
MailHandler	✓		✓ by means of VipEMailBean	
ObjectHandler	✓	✓	✓ by means of VipObject HandlerBean	✓ by means of VipWebService_Port
ObjectLoader	✓	✓		
SystemHandler	✓	✓		
WorkflowHandler	✓		✓ by means of VipWorkflow Bean	✓ by means of VipWebService_Port

Functionality of the	WCM Java API	Remote API	Portal Manager API	WCM WebServices
`DeploymentHandler`	✓	✓		
`ContextHandler`	✓	✓		
`LivelinkAdmin Handler`	✓		✓ by means of `LivelinkUser Bean`	
`LivelinkObject Handler`	✓		✓ by means of `LivelinkObject Bean`	

Typographic Conventions

The following conventions are used in the text to draw attention to program elements, etc.:

Item	Font or Symbol	Examples
Program interface, such as menu commands, windows, dialog boxes, field and button names	*Menu → Entry*	*File → Create*
Paths to directories, file and directory names	**Drive:\Directory\File name**	**D:\WCM\ admin.bat**

Quotations from program code or configuration files	`Code quotations`	`<head>`
		`<title>heading </title>`
		`</head>`

Variables, i.e. placeholders for specific elements	{variable}	{WCM installation directory}

Important **information** and **warnings** are enclosed in gray boxes. Make sure to read such information to avoid losing data or making errors when using and managing WCM systems.

CHAPTER 2
Implementation Basics

This chapter introduces the architecture of the WCM Java API. It explains how to integrate server agents developed on the basis of the WCM Java API into the WCM system. Moreover, general implementation principles, such as logging in and out of the WCM system, context management, and error handling, are described.

Highlighted topics include:

- structure of the WCM Java API (see section “Architecture of the WCM Java API” starting on page 20)
- accessing the WCM Java API functions via server agents: integrating server agents in the WCM system as well as starting and stopping agents (see section “Server Agents” starting on page 22)
- accessing the WCM Java API functions via the remote API (see section “Remote API” starting on page 30)
- logging in and out as well as managing contexts (see section “Authentication and Contexts” starting on page 36)
- processing errors (see section “Error Handling” starting on page 40)
- localizing messages (see section “Localized Messages” starting on page 42)
- managing custom connections, e.g. to third-party systems (see section “Pool Management” starting on page 44)

Architecture of the WCM Java API

The central class of the WCM Java API is the class `VipRuntime`, the point of access to all WCM Java API interfaces.

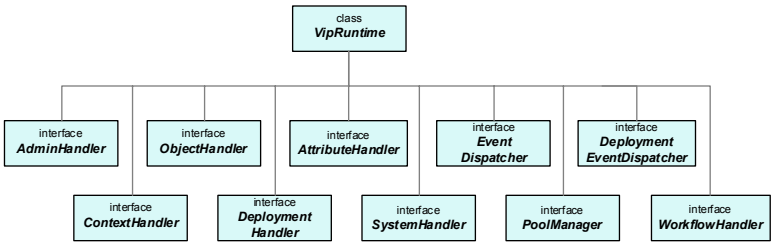


Fig. 1 – Basic WCM Java API interfaces

The individual interfaces offer the following functionalities:

- The `ContextHandler` interface: the context management interface provides methods for logging into and out of the current server, changing the password, or retrieving the profile of a logged-in user.

  See section “Authentication and Contexts” starting on page 36

- The `PoolManager` interface: this interface provides access to connections that are based on custom connection types (e.g. for integrating third-party systems). These types of connections can be managed via the pool mechanism of Livelink WCM Server.

  See section “Pool Management” starting on page 44

- The `AdminHandler` interface: this interface allows access to the data and functions of the user and system administration of Livelink WCM Server. This enables you to create and change principals, edit assignments, request information on servers, websites, and deployment systems, as well as to retrieve and change run levels.

See chapter 3 “User Administration” and chapter 6 “System Information and Administration”

- The `EventDispatcher` and `DeploymentEventDispatcher` interfaces: the event management offers an interface for registering (and deregistering) event listeners. This interface is a central element for the current server and dispatches events to registered listeners. These events may refer to the entire WCM system, websites, or deployment systems.

  Agents can register for an event which is triggered before an action is executed (`PrepareEvent`) or after the action has been executed successfully (`Event`). In the first case, an action can be aborted by a veto (in the form of an `Exception`).

  See chapter 4 “Event Processing”

- The `ObjectHandler` interface: the object management interface permits access to the WCM objects of a website. All known Content client actions that can be performed on WCM objects are supported (e.g. add, submit, release, filter, and delete WCM objects).

  See section “Actions of the `ObjectHandler` Interface” starting on page 108

- The `AttributeHandler` interface: this interface can be used to read all attribute sets and object categories used for a website.

  See section “Editing Special Attributes” starting on page 128

- `The WorkflowHandler` interface: The interface for managing workflow objects allows you to assign workflows to WCM objects, remove this assignment, and forward WCM objects in the workflow.

- The `DeploymentHandler` interface: this interface allows access to the metadata of pages, such as path and URL to the generated file. It also offers information on the status and errors of the deployment.

  See section “Information on the Deployment” starting on page 164

- The `SystemHandler` interface: this interface provides access to the system administration functions. These include reading server reports and logs, as well as tracing.

  See chapter 6 “System Information and Administration”

Server Agents

Server agents are used to access the Livelink WCM Server functions via the WCM Java API. You can also use the so-called *remote API* to develop programs, which are executable outside a WCM server. The remote API is described in section “Remote API” starting on page 30.

Server agents are Java implementations on the basis of the WCM Java API. Server agents are loaded when a server starts. They are executed in the same Java Virtual Machine (JVM) as the respective server. For this reason, all agents must be developed on the basis of the Java SDK (Software Development Kit) used for the execution of the servers.

The following figure illustrates the implementation on the basis of server agents

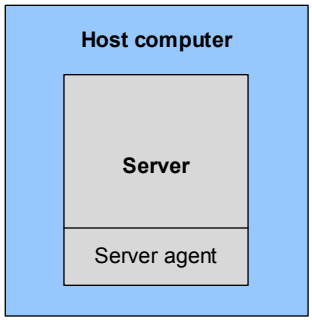


Fig. 2 – Implementation of a server agent

Note: To ensure correct loading of a server agent, compress the respective classes in a JAR file and copy this file to the directory **\external_lib** in the WCM installation directory. Restart the respective server. If the server on which the agent is to be executed runs as a web application in an application server, copy the JAR file also to the directory **{web application directory}\Web-Inf\lib**.

Configuring Parameters and Constructors

Server agents can be integrated and configured in the Admin client via *Configuration → Server agents*.

Here, you define the required properties of the agent and set the according values. The properties are defined by simple parameters to which values are assigned in the Admin client. It is also possible to define tree-like hierarchies of parameters. For this purpose, you create nodes. A node can contain individual parameters or additional sub-nodes.

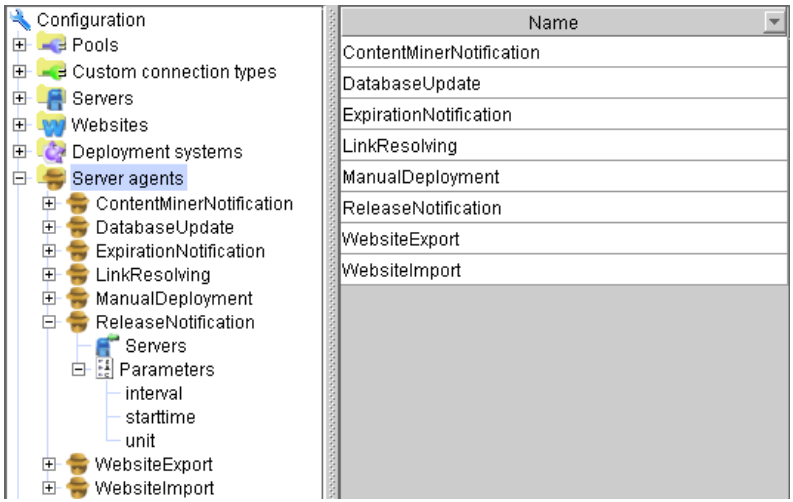


Fig. 3 – Managing server agents in the Admin client

Each server agent must implement the `ServerAgent` interface (package: `de.gauss.vip.api`). The agent always requires a constructor declared as `public`. This constructor must expect either

- a parameter of type `java.util.Properties`

  or

- a parameter of type `de.gauss.vip.config.PropertyMap`.

  The `de.gauss.vip.config.PropertyMap` interface is described in the Javadoc on the WCM Java API.

If the parameters are not organized in a flat structure, but in a hierarchy of nodes, you must use an argument of type `de.gauss.vip.config.PropertyMap`. The constructor should not perform any time-consuming tasks. Otherwise, it would block the calling server thread.

When the agent object is created, the respective constructor is called with the properties set in the configuration. The method `getProperty` or `getPropertyValue` reads the properties. The settings of an agent also contain the Java class of the agent which must be accessible via the class path.

Example 1 – Agent with hierarchical arguments

The following figure shows the configuration of an agent whose arguments are summarized in nodes.

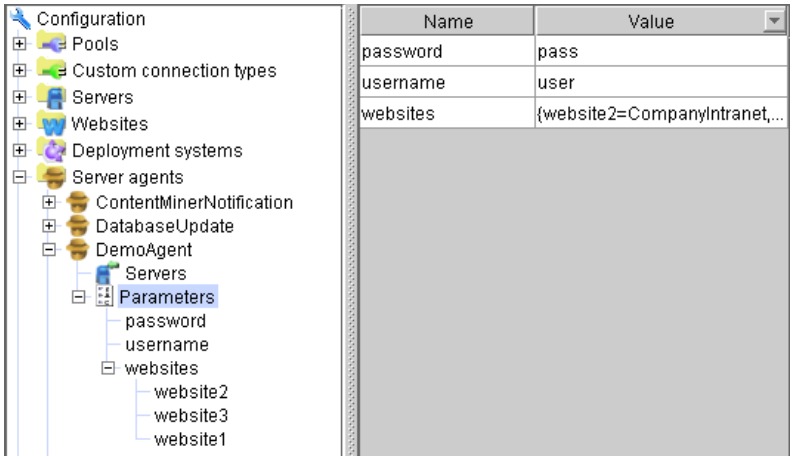


Fig. 4 – Configuring the demo agent with nodes

The constructor of the `DemoAgent` class with nested parameters looks as follows:

```
public class DemoAgent
    implements de.gauss.vip.api.ServerAgent
{
    [...]
    private final String username;
    private final String password;
    private final String website1;
    private final String website2;
    private final String website3;

    public DemoAgent(de.gauss.vip.config.PropertyMap config)
        throws de.gauss.vip.config.KeyNotFoundException
    {
        de.gauss.vip.config.PropertyMap websites
            = config.getPropertyMap("websites");
        username = config.getPropertyValue("username");
        password = config.getPropertyValue("password");
        website1 = websites.getPropertyValue("website1");
        website2 = websites.getPropertyValue("website2");
        website3 = websites.getPropertyValue("website3");
```

```
    }
    [...]
}
```

Example 2 – Agent with arguments without nodes

The demo agent without nodes might be configured as follows:

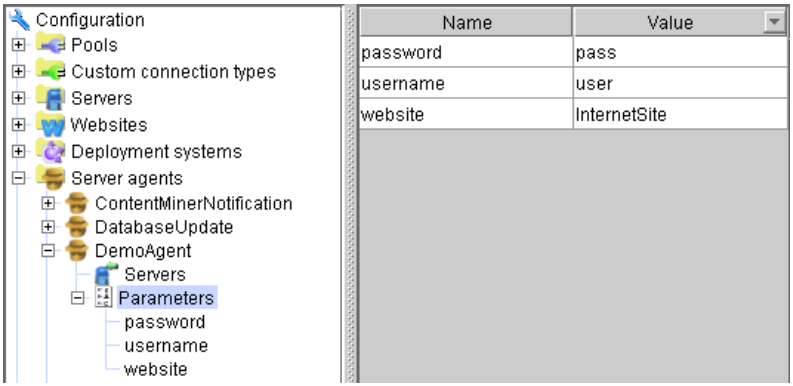


Fig. 5 – Configuring the demo agent without nodes

The constructor of the `DemoAgent` class looks like this.

```
public class DemoAgent
    implements de.gauss.vip.api.ServerAgent
{
    [...]
    private final String userName;
    private final String password;
    private final String website;

    public DemoAgent(java.util.Properties config)
    {
        userName = config.getProperty("username");
        password = config.getProperty("password");
        website1 = config.getProperty("website");
    }
```

```
    [...]
}
```

Additional Information on Server Agents

For each server agent, several entries must be defined that offer more detailed information about the agent:

- manufacturer, description, and version

  These values are read by means of the methods `getManufacturer`, `getDescription`, and `getVersion`. These entries are, for example, displayed in the Admin client under *System administration → Active servers → {server name} → Reports → ServerAgentManager*.

- required version of Livelink WCM Server

  These values are read by means of the methods `getRequiredVIPMajorVersion` and `getRequiredVIPMinorVersion`. They define the Livelink WCM Server version required by the agent (e.g. version 8.6 is specified by 8 for `Major` and 6 for `Minor`). If the product version running on the server is lower than the version returned by the agent, the server will not start the agent.

Starting Server Agents

Server agents are started by means of the `start` method of the `ServerAgent` interface. The server calls this method in the following situations:

- during startup, i.e. when switching from run level `AGENTS_STOPPED` to run level `SINGLE_USER` (see also section “Retrieving and Setting Run Levels” starting on page 173)

  This has some implications as the server has not completed booting at this point. In particular, it is not possible to access any website at this time.

- when explicitly starting the agent via the Admin client

Before each start of the agent, a **new** instance of the agent class is created (by using the constructor of this class, see above). Afterwards, the `start` method of the agent instance is started in a separate thread. This thread terminates when the call of the `start` method returns. Depending on the actions of the agent, this may be very shortly or a long time after the call.

> **Note:** When implementing the `start` method, make sure that no time-consuming actions are performed as this would delay the booting process of the server. In the `start` method, it is not possible to wait for the servers to boot. Instead, it is advisable to only register for the respective event, e.g. `SERVER_UP` or `WEBSITE_UP`, in the method.

In the `start` method, the agent can register for events fired by the server in the case of status changes (referring to the system or to individual WCM objects). If the agent starts threads of its own, these threads can only be terminated by the agent itself. In this case, the `start` method must return the value `true`. This guarantees that the `stop` method of the agent is called if required in order to terminate the started threads.

For an example of starting and stopping agents, refer to section “Monitoring Deployment Events” on page 215.

Note: The return value of the `start` method indicates whether the agent has been started successfully. If the `start` method returns the value `false`, the Admin client immediately displays the agent as stopped.

Stopping Server Agents

Server agents are stopped by means of the `stop` method of the `ServerAgent` interface. This method should be used to release bound resources. It is called when

- the `start` method has returned the value `true` (or if it has been called, but not yet terminated)

  and either

- the server changes from run level `WEBSITE_INACCESSIBLE` to run level `AGENTS_STOPPED`

  or

- the agent is stopped by means of the Admin client

Notes

The return value of the `start` method determines whether the `stop` method is called at all. If `start` returns the value `false`, the `stop` method is never called.

The `stop` method should not perform any time-consuming tasks. Otherwise, it would block the calling server thread.

Notification of Events

Events are fired for all actions in the WCM system that change the status of WCM objects or system components. Usually, actions of a server agent are triggered by an event, e.g. adding an object or booting a website.

To be informed about events, a server agent must register a so-called *EventListener* with an *EventDispatcher*. The agent is informed about the respective event as long as it is registered with the event dispatcher. Thus, notification is independent of the status of the agent itself. It depends exclusively on the registration.

For detailed information on events and how they are processed, refer to chapter 4 “Event Processing” and to section “Monitoring Deployment Events” on page 215.

Remote API

The remote API enables you to develop Java programs on the basis of the WCM Java API. In contrast to server agents, these programs can be executed outside a WCM server. The programs developed this way can be executed on any computer that can connect to a WCM server. Via the remote API, you can use the majority of the WCM Java API interfaces – only the way they are called is different.

The remote API has been implemented on the basis of Java's RMI mechanism (remote method invocation). The following figure illustrates the implementation of a Java program that addresses a server via the remote API.

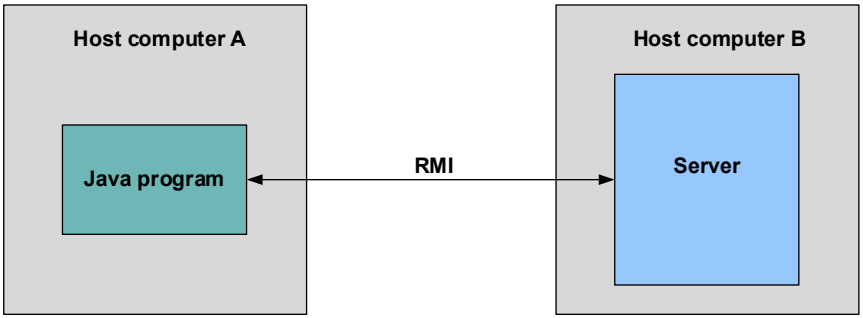


Fig. 6 – Implementation on the basis of the remote API

Basic Components of the Remote API

> **Note:** For using the remote API, you must first activate the remote API functionality for the desired server in the Admin client. For more information, refer to the Livelink WCM Server Administrator Manual.

In order to provide the remote API, two interfaces were introduced in the WCM Java API:

`Runtime` interface

The interface `de.gauss.vip.api.Runtime`. This interface provides almost the same methods as `VipRuntime`. For detailed information, refer to the Javadoc. The `Runtime` interface constitutes the basis for the other interfaces.

`RemoteRuntime` interface

The interface `de.gauss.vip.api.RemoteRuntime`. This interface extends the `Runtime` interface and offers functions for establishing a connection to a server.

This interface can be seen as the counterpart of the VipRuntime interface of the WCM Java API. Most interfaces of the WCM Java API that can be requested via VipRuntime are also available via RemoteRuntime. Exceptions are the interfaces of the following packages:

- de.gauss.vip.api.wfm
- de.gauss.vip.api.pool
- de.gauss.vip.api.mail

The RemoteRuntime interface is requested via the getRuntime method of the VipRuntime interface.

```
public RemoteRuntime getRuntime(String host, String server,
                                int port)
```

Three parameters must be supplied:

- the name of the computer hosting the server that is to be addressed via the remote API
- the name of the desired server
- the port of the server for the remote API connection. By default, port 1099 should be used.

Example

In the following code sample, the Runtime interface is requested by the server “Master_Content”. Afterwards, a login is performed.

```
...

RemoteRuntime rtEdit = VipRuntime.getRuntime (
                  "wcmserver.company.example", "Master_Content", 1099);
ContextHandler ctxHandler = rtEdit.getContextHandler();
ContextId cid = ctxHandler.login("admin", "admin" );
...
```

If the remote API connection to the server is to be established via secure connections (SSL), you can use the `getRuntime SSL` method instead of `getRuntime`. An SSL-secured `RemoteRuntime` interface is called as follows:

```
public RemoteRuntime getRuntimeSSL(String host, String server, int port,
                                   String clientTrustStoreLocation)
```

In addition to the parameters that must be specified for the method `getRuntime`, you must also specify the path to the truststore to be used for verifying SSL certificates. This truststore must be located on the local computer and contain the certificate used by Livelink WCM Server.

Notes:

For information on adding the WCM certificate to a truststore, refer to the Livelink WCM Server Installation Manual.

Operation of the remote API via SSL is independent of the communication settings for the respective server. It may, for example, be possible that remote API connections run via SSL, whereas the other connections of the server are not secure.

Requirements for Developing a Remote API Application

To develop an executable Java program by means of the remote API, you require the following files from the WCM installation directory on the computer on which the Java program is executed.

- From the subdirectory **\lib**:
  - **jcert.jar**
  - **jnet.jar**

- **jsse.jar**
- **vipcore.jar**

- From the subdirectory **\config**: the complete subdirectory **\resource**. This directory contains the properties files with the messages and GUI texts of Livelink WCM Server required for starting the Java program.

  Create a directory on the local computer and copy the directory structure **\config\resource** to this directory. In the class path of the Java program, there must be a reference to the new directory.

Example

The following code sample is part of a simple remote API program:

```
package .....;

import com.sun.net.ssl.internal.ssl.Provider;
import java.security.Security;

import de.gauss.vip.api.admin.ContextHandler;
import de.gauss.vip.api.lang.ContextId;
import de.gauss.vip.api.VipRuntime;


public class RemoteTestClient
{
      public static void main(String[] args)
      {
         System.out.println(" starting RemoteTestClient " );
         try
         {
            RemoteTestClient test = new RemoteTestClient();
            test.start( args); //arg[0]=hostname, arg[1]=servername,
                     //arg[2]=port, arg[3]=secure("true"/"false")
         }
         catch ( de.gauss.vip.api.exception.VipApiException vae )
         {
            vae.printStackTrace();
         }
         catch ( Throwable t )
         {
            t.printStackTrace();
```

```
    }
    System.out.println(" RemoteTestClient stopped  " );
}

public RemoteTestClient()
{
}

public void start(String[] args) throws Exception
{
    //hostname
    String host = args[0];
    //name of the WCM server
    String serverName = args[1];
    //the remote port of the WCM server
    int remotePort = Integer.parseInt(args[2]);
    //flag indicating whether to use a secure connection
    boolean secure = Boolean.getBoolean(args[3]);
    //reference to API implemenation
    de.gauss.vip.api.RemoteRuntime runtime = null;
    if ( secure ) //initialize ssl
    {
        //set up location of trust store in the file system....
        String trustStoreLocation =
                            "d:\\wcm\\keys\\gauss_vip80.trust";

        //Create remote API implementation instance ..
        runtime = VipRuntime.getRuntimeSSL( host, serverName,
                                  remotePort, trustStoreLocation);
    }
    else
    {
        //Create remote API implementation instance ..
        runtime = VipRuntime.getRuntime( host, serverName,
                                                    remotePort );
    }
    //start using remote API .....
    ContextHandler ctxHandler  = runtime.getContextHandler();
    ContextId cid = ctxHandler.login("..." , "...");
    .......
```

Authentication and Contexts

Access to a WCM system via the WCM Java API requires user authentication in order to be able to use the methods of the individual interfaces, especially object management and administration. For many actions, the access control lists of the WCM objects are checked to determine whether the user has the required rights.

The `ContextHandler` interface is used to log in to or log out of a server. The methods of this interface provide the following functions:

- authenticating a user in the WCM system
- managing the context of a user login
- changing the password
- determining the user belonging to a context (including the user's profile with all websites and functional areas directly and indirectly assigned)
- logging in as the substitute for another user

A context usually has a limited life span. If no actions are performed over a certain period of time, the context becomes invalid. The WCM Java API provides methods for using a context for any period of time even if no actions are performed.

In addition to user login functions, the `ContextHandler` interface provides two predefined contexts that can be used for accessing WCM objects: World and Backup (see section "Predefined Contexts" on page 38).

> **Note:** In addition to user contexts, there are transaction contexts (see "User and Transaction Contexts" on page 117).

Login and Logout

Users may log in as often as they like. Each login returns a different `ContextId` object which is unique within the WCM system.

A `ContextId` object hides the actual implementation of the user context, as well as the assigned information (which user logged in when and where, which actions the user performed, etc.). For security reasons, the interface does not contain any methods.

Using the `ContextHandler` interface, you can log in as the substitute of another user and thus work with the object rights of this user. The following example demonstrates a substitute login:

```
ContextId cid;
ContextHandler ch;
ch = VipRuntime.getContextHandler();
cid =ch.login("herbert","vip");
ch.substituteLogin(cid,"jstein ");
// --- Executing actions with the cid
...
//---Log out
ch.logout(cid);
```

After logging in as a substitute, the existing `ContextId` represents the context of the substitute and no longer the context of the originally logged-in user.

For each logged-in user, the profile can be retrieved. The following call returns a `User` object which automatically contains the profile.

```
ContextId cid = VipRuntime.getContextHandler().login(userName,
                                                     password);
User user = VipRuntime.getContextHandler().getUser(cid);
System.out.println("hasProfile() returns ="+user.hasProfile());
```

Notes:

The system administration offers a detailed overview of the users that are currently logged in to the WCM system. The `getActiveUsers` method of the `SystemHandler` interface is used for this purpose (see section “Retrieving Logged-in Users” on page 160).

Every user with write access to the WCM system uses up a license. This license is released after one minute at the earliest, even if the user logs out before this minute has elapsed. When the minute has elapsed, the license will be released together with the user logout.

Predefined Contexts

Livelink WCM Server provides two predefined contexts which can be used by server agents for accessing WCM objects without logging in:

- World – This context represents a login with the rights for the group “World”. The group’s rights in the website are determined by the website’s access control lists. The `getWorldContextId` method returns this context, which can usually read only certain WCM objects.
- Backup – This context represents a user that has read access to all WCM objects of a website. This makes this context especially suited for programming backup agents. The `getBackupContextId` method is used for this purpose.

Both of the contexts do not need to be refreshed (see the following section) and do not have a user profile.

Refreshing Contexts

The context of a logged-in user can be retrieved by means of the methods `login` or `substituteLogin`. Context have only a limited life span if no actions are performed within a certain period of time. After a certain period of time without server access, the context becomes invalid. You can configure this time span (*Expiration interval*) in the Admin client.

To manually refresh a user context, use the `refreshContextId` method. After this call, the refreshed context remains valid for the specified expiration interval.

The user context can be automatically refreshed in a separate thread. This “Refresh” thread is activated using `startContextRefresh`. You can specify the refresh time interval. If it is not specified, half the expiration interval is used.

The `stopContextRefresh` method stops the thread belonging to the `ContextId`. Especially when performing actions triggered by received events, it is necessary for the user context to be automatically refreshed if the actions are to be executed in a common user context. The following example demonstrates this:

```
public class DemoAgent
    implements de.gauss.vip.api.ServerAgent
{
    [...]
    private de.gauss.vip.api.admin.ContextHandler cxtHandler;
    private de.gauss.vip.api.lang.ContextId contextId;

    private final String userName;
    private final String password;
    [...]
    public DemoAgent(java.util.Properties config)
    {
        userName = config.getProperty("username");
        password = config.getProperty("password");
        [...]
    }

    public boolean start(String serverType, int majorVersion,
```

```
                        int minorVersion, String patchLevel)
    {
        try
        {
           cxtHandler = de.gauss.vip.api.VipRuntime.getContextHandler();
            contextId  = cxtHandler.login(userName, password);
            cxtHandler.startContextRefresh(contextId);
        }
        catch (de.gauss.vip.api.exception.VipApiException vax)
        {
            vax.printStackTrace();
            return false;
        }
        [...]
    }

    public void stop()
    {
        [...]
        try
        {
            cxtHandler.stopContextRefresh(contextId);
            cxtHandler.logout(contextId);
        }
        catch (de.gauss.vip.api.exception.VipApiException vax)
        { /* ignored */ }
    }
    [...]
}
```

Error Handling

The WCM Java API uses the Java exception mechanism to report errors in Livelink WCM Server. Any method that outputs an error throws an exception in the WCM Java API.

The basic class for error handling is the class `VipApiException`. In addition, the WCM Java API provides several exception classes that derive from `VipApiException`. These exceptions can be thrown due to errors that occur during the execution of a method. These derived exception

classes include, for example, `AccessDeniedException` (if an action is not permitted because of the access rights defined for the object) or `LoginException` (if login to the WCM system failed). For detailed information on the individual exceptions, refer to the Javadoc.

Localized Error Messages

The following methods are available for returning localized texts for an exception:

- `getMessage`: the method provides a localized text that describes the cause of the error.
- `getMessages`: the method returns all localized messages concerning the error as a string array.

Example

If submitting an object fails, the following messages could be returned via `getMessages`:

- First message: “Could not submit object 4711.”
- Second message: “The parent topic has not yet been submitted.”

In this case, the second message indicates the actual cause of the error. The first message describes the result, i.e. that the object could not be submitted.

Localized Messages

If you want your application to use the same localization mechanism as Livelink WCM Server and to output translated messages depending on the language settings of the logged-in user, you can use the functions of the basic class `UserMessage` (`de.gauss.vip.api.UserMessage`). The most important method of this class is `getString(locale)`. This method returns a representation of the message in the specified language (without considering the country code).

A message always consists of a key in the form of a language-independent character string and, optionally, several arguments (arbitrary Java objects). The translation of the language-independent key is read from a message file (***.properties**). Every message class has a file containing an assignment of message keys to message texts. For each supported language, a separate file exists.

> **Note:** The language-independent key must be entered in the message file **exactly** in the same way as it is used in the code.

The message files are located in the following subdirectories of the WCM installation directory:

- **\config\resource**
- **\admin\config\resource**
- **\contentminer\config\resource**

When using an application server, the message files are also located in the following directories:

- **{WCM installation directory}\admin\web\WEB-INF\classes\config\resource**
- **\{web application directory}\WEB-INF\classes\config\resource**

The file names have the following format:

<className>_<langCode>.properties.

The language code (<langCode>) in the file names of the message files complies with ISO 639 (a list of language codes according to ISO 639 can be found at http://www.w3.org/WAI/ER/IG/ert/iso639.htm). The file for user-defined messages in English thus has the following name:

UserMessage_en.properties.

To make the localized messages available on the different servers of the WCM system, the respective configuration files must be copied to the server's directories provided for this purpose.

Example

The code contains the following (language-independent) key for a simple welcome text:

"USER_WELCOME"

This message has two arguments: "{0}" and "{1}". For both arguments, the toString method is called.

In the **UserMessage_en.properties** properties file, the English message text for the key is listed as follows:

```
...
USER_WELCOME=Hello, {0}. Welcome to ''{1}''.
...
```

If the current user has the ID "jstein" and is logged in to the "InternetSite" website, the following code returns the message: "Hello, jstein. Welcome to 'InternetSite'."

```
UserMessage msg = new UserMessage("USER_WELCOME",
                                                  userName, webSiteName);
System.out.println(msg.getString(Locale.en_US));
```

Pool Management

In a WCM system, numerous connections are used. The WCM servers communicate with each other, connections to the database or the LDAP directory service are required, and integration with a third-party system may be necessary. All connections of a WCM system are managed in pools.

Each pool is based on a connection type that characterizes the type of connections in this pool. There are JDBC pools for the connections to the database and WCM pools for the connections between the WCM servers. The predefined connection types are called default connection types. Connections on the basis of these connection types are managed by the internal `PoolManager` of Livelink WCM Server.

You can create and manage custom connection types. On the basis of custom connection types, you can create pools for custom connections, e.g. for integrating third-party systems with the WCM system. Like the pools on the basis of standard connection types, custom pools are managed via the pool mechanism of Livelink WCM Server.

For detailed information on pools and connection types, refer to the Livelink WCM Server Administrator Manual.

WCM Java API Components for Custom Connections

The following WCM Java API components support the creation and administration of connections based on custom connection types:

- The class `de.gauss.vip.api.pool.PoolConnection`: connections based on custom connection types must be derived from this class. In this way, they can be managed by the pool mechanism of Livelink WCM Server. In the following, these connections are called custom connections.

  The methods `close`, `isValid`, `execute`, and `customOpen` of the `PoolConnection` class must be programmed completely in a derived class.

- The `de.gauss.vip.api.pool.PoolManager` interface, which encapsulates the internal `PoolManager` of Livelink WCM Server. The `PoolManager` manages the connections in pools, i.e. it establishes and closes the connections on demand. The connections in these pools must be derived from the `PoolConnection` class. The pools must have unique names. Use the Admin client to create and edit pools.

  If an agent requires a custom connection, it can use the `getConnection` method of the `PoolManager` interface to request the connection. The pool name must be specified when calling this method. When the connection is no longer required, it **must** be returned to the pool by means of the `freeConnection` method. If the connection is not returned, it is regarded as open, even if it is not used. Thus, it cannot be used for other actions.

  The `PoolManager` interface is accessed via `VipRuntime.getPoolManager`.

Procedure for Setting up Custom Connections

When developing an agent that is to establish a custom connection to a third-party system, proceed as follows:

1. Create a class for the connection – derived from the basic class `PoolConnection`.

   Program the methods `close`, `isValid`, `execute`, and `customOpen` according to the requirements of the third-party system. If, for example, a port is required for establishing the connection to the third-party system, this value can be supplied in the method `customOpen`.

   If you want to make it possible to request a report for the user-defined pool via the Admin client or the `SystemHandler` interface, the `getReport` method must additionally be implemented (see “Reports for Custom Connections” on page 47).

2. In the Admin client, create a custom connection type for connections to the third-party system.

   As *Class name*, enter the name of the connection class that you have derived from the basic class `PoolConnection`. Depending on the third-party system, additional parameters may have to be defined.

3. Create a new pool on the basis of this connection type. This pool contains the connections to the third-party system and is managed by the `PoolManager`.

4. The agent that is to establish the connection to the third-party system, must call the `PoolManager` method `getConnection` to get a connection from the user-defined pool. The `PoolManager` establishes the connection to the third-party system by means of the `customOpen` method. In this process, the parameters defined for this pool in the Admin client are supplied.

Reports for Custom Connections

For the various software components of the WCM servers, reports are created automatically. These reports log the actions in the WCM system. You can view the reports in the Admin client (see Livelink WCM Server Administrator Manual).

The reports can also be accessed via the WCM Java API. The `SystemHandler` interface provides the method `public List getCoreReport` for requesting a specific report from a server. For detailed information, refer to section “Server Reports” on page 175.

Information on custom connections is provided in a corresponding report of the CustomPoolManager, which requests the necessary information from the managed custom connections. This information is only collected if the method `public List getReport(String prefix)` is implemented in the corresponding custom connection.

The method returns a list. This list must be filled with string objects representing the desired report data. Each string in the list corresponds to a line in the CustomPoolManager report for this custom connection. The supplied string `prefix` represents an indentation. It is usually filled with spaces.

The respective report can be called via the `SystemHandler` or the Admin client.

Example

The following code sample shows how the `getReport` method is programmed in the custom connection “TestConnection”. The method writes the name of the computer to which the connection has been established and the connection status to the list. The filled-in list is returned.

```
public List getReport(String prefix)
{
  List report = new ArrayList();
  report.add(prefix+"TestConnection : ");
  report.add(prefix+"connected with " + getConnectionPeerName() );
  report.add(prefix+"connection status : " + getConnectionStatus() );

  //empty line..

  report.add("");
  return report;
}
```

CHAPTER 3

User Administration

The WCM Java API provides access to the user administration functions of Livelink WCM Server. For this purpose, the `AdminHandler` interface offers the following functions:

- adding and deleting principals, retrieving and editing profiles of principals, searching principals (see section “Editing Principals” on page 54)
- assigning WCM attributes to LDAP principals, thus allowing them access to the WCM system (see section “Setting WCM Attributes in LDAP” on page 61)
- editing principal rights (see section “Editing Rights of Principals” on page 64)
- adding and deleting functional areas as well as retrieving information on functional areas (see section “Editing Functional Areas” on page 71)
- assigning users to groups/roles, assigning functional areas and websites to principals; removing these assignments (see section “Editing Principal Assignments” on page 74)

An instance of the administration interface can be created via `VipRuntime.getAdminHandler`.

Note: For accessing the user administration data, administration rights are required, e.g. the right “Create, change, delete principal” for adding principals and editing principal information. For an overview of all administration rights, refer to table “The individual administration rights” on page 66.

The following sections briefly describe the administration interface. Unless otherwise specified, all classes and interfaces are included in the package `de.gauss.vip.api.admin`.

Accessing Livelink users

If Livelink WCM Server uses the Livelink user administration, you can use the `AdminHandler` interface of the WCM Java API to access the users, in the same way as for user administrations based on an RDBMS or an LDAP system. Additionally, the WCM Java API provides the `LivelinkAdminHandler` interface for accessing Livelink groups and users. This interface wraps the user management functions of the Livelink API (LAPI for short). In this way, you can access Livelink principals, even if the WCM system does **not** use the Livelink user administration.

For information on accessing the Livelink user administration via the `LivelinkAdminHandler` interface, refer to section “Livelink User Administration” on page 183.

The `Principal` Interface

The Administration server manages users, groups, and roles, among other things. In the WCM Java API, these “right bearers” are represented by objects that implement the `Principal` interface. Each principal has a unique name and a type. The `getUser`, `getGroup`, and `getRole` methods of the `AdminHandler` interface return a corresponding `Principal` object.

The `TYPE_GROUPROLE` type represents a group-role that can be created by an object of the `GroupRole` class. Group-roles are not managed in the Admin client, they are provided for use in access control lists of WCM objects. A group-role defines the intersection of the users of a group with the users of a role. The group-role includes those users who are assigned to both the group and the role (please refer to the Content client User Manual for further information).

The `TYPE_WORLD` type represents the predefined group “World” and is also used only for the definition of access rights in the access control lists of WCM objects. In a WCM system, there is only one object of the respective `World` class (singleton pattern).

For the user administration, the `TYPE_USER`, `TYPE_GROUP`, and `TYPE_ROLE` types are particularly relevant. These types have three associated classes: `User`, `Group`, and `Role`. Users are identified by a user ID, while groups and roles are identified by a unique name. The further properties (“Profile”) are retrieved via the `AdminHandler` interface. Each of the three classes provides a `hasProfile` method which checks whether the other properties exist.

Editing Principals

The administration interface provides methods for editing users, groups, and roles.

This section explains the methods for the following purposes:

- adding principals (see “Adding Principals” on page 54)
- editing principal settings (see “Profiles of Principals” on page 56)
- searching principals (see “Searching Principals” on page 58)
- deleting principals (see “Deleting Principals” on page 61)

Note: For editing and deleting principals, it does not matter whether the user data is stored in an RDBMS, in Livelink, or an LDAP directory service. Whether it is possible to add principals depends on the LDAP directory service used (see Livelink WCM Server Installation Manual).

Adding Principals

Principals are added by means of the `PrincipalFactory`. The factory is only used for creating principal objects. To persistently add the new users, groups, or roles to the WCM system, the `add` method of the `AdminHandler` interface must be called afterwards.

Use the `getPrincipalFactory` method of the `AdminHandler` interface to get an instance of `PrincipalFactory`.

When a principal is added by means of one of the methods `createUser`, `createRole`, or `createGroup`, various parameters with the properties of the new principal are supplied.

Users, groups, and roles have a number of common properties that can be set when they are added:

- a name (`cn`)
- an LDAP position (`ldapPosition`) – only if an LDAP directory service is used for storing the user data. If an RDBMS or Livelink is used, `null` must be supplied.
- an e-mail address (`email`)

Additionally, the following properties have to be set for users:

- the user ID for logging in to the WCM system (`userID`)
- the password for login (`vipUserpassword`)
- a flag indicating whether the password must be changed when the user logs in to the WCM system for the first time (`initPassword`)
- the user's preferred language (`language`)

Note: The permission to access the WCM system (`vipAccess`) is implicitly granted when the principal is added.

The following example adds a user to an LDAP user administration:

```
// login with appropriate administration rights
ContextId cid = VipRuntime.getContextHandler().login(...);

// define the LDAP position of the new user
String ldapRootPos = "ou=admin, o=test, c=de";

// define the name (cn) and ID of the new user
String userCN = "Joshua Stein";
String userId = "jstein";

//define passwort, e-mail address, and language of the new user
String passwd = "4711";
String email = "joshua.stein@company.example";
LocaleValue loc = new LocaleValue("en_US");

// create the new user by means of the PrincipalFactory
AdminHandler aH = VipRuntime.getAdminHandler();
PrincipalFactory princFac = aH.getPrincipalFactory(cid);
User usrNew = princFac.createUser(userId, userCN, ldapRootPos,
```

```
                email, loc, passwd, true);

//add the new user to the WCM system by means of the AdminHandler
// method void add(ContextId, Principal)
aH.add( cid, usrNew );
```

Profiles of Principals

Each principal has a number of properties which constitute this principal's profile. Users, groups, and roles have a number of common properties, such as name and e-mail address. For groups and roles, the users belonging to the group/role can be specified. Users have additional properties, such as the user ID and the password for logging in to the WCM system, a language, and a substitute.

Notes:

Not all of a principal's properties can be changed subsequently, e.g. the user ID or the name of a group/role.

For information on changing the administration rights and the default object rights of principals, refer to section “Editing Rights of Principals” on page 64. For notes on editing the assignments of principals, refer to section “Editing Principal Assignments” on page 74.

The functions of the WCM Java API can be used to read and edit the profiles of principals. The interfaces `User`, `Group`, and `Role` supply methods for this purpose.

Retrieving Profiles

To retrieve the profile of a user, group, or role, use the `getUserProfile` method of the `AdminHandler` interface.

The following example determines the profile for the user with the ID “jstein”. In this example, the profile also contains all indirectly assigned functional areas and websites, i.e. the functional areas and websites of jstein’s roles and groups.

```
AdminHandler ah = VipRuntime.getAdminHandler();
ContextId cid = VipRuntime.getContextHandler().login("admin", password);
User user = ah.getUserProfile(cid, "jstein",
            AdminHandler.INDIRECT_ASSIGNMENTS);
List websites = user.getWebsites();
Iterator i = websites.iterator();

while (i.hasNext())
{
   Website w = (Website)i.next();
   System.out.println("User '"+user.getName()+
                      "' is assigned to website '"+w.getName()+"'.");
}
```

Editing Profiles

In order to modify a profile, it must be loaded first. Use the appropriate methods of the `User`, `Group`, or `Role` interface to set the desired properties. To persistently store the changes in the user administration, call the `update` method of the `AdminHandler` interface at the end.

The following example changes the e-mail address and the language setting of the user with the ID “jstein”.

```
// login
ContextId cid = VipRuntime.getContextHandler().login(...);

// load user profile of "jstein"
AdminHandler aH = VipRuntime.getAdminHandler();
User usrJOS = aH.getUserProfile(cid, "jstein",
AdminHandler.INDIRECT_ASSIGNMENTS );

//set new e-mail address and locale for the user
usrJOS.setLocale(new LocaleValue("en_US") );
usrJOS.setEMailAddress(jstein@company.example);
```

```
//save changes in the user administration
aH.update(cid, usrJOS);
```

Searching Principals

By means of the `getPrincipals` method of the `AdminHandler` interface, principals meeting a certain criterion can be retrieved. The search criterion is represented by an object of the type `Filter`.

> **Note:** For detailed information on the `Filter` class, refer to section “Searching WCM Objects” starting on page 133.

The principals found do not have a profile. The profile must be loaded subsequently by means of the methods `getUserProfile`, `getGroupProfile`, or `getRoleProfile`.

The `SearchableKeys` interface provides constants that can be used as search criteria. The following table gives a short overview of the search criteria and some examples:

Table 2 – Criteria for searching principals and examples

Search criterion	Constant	Example
User ID	`USER_ID`	Searching users whose ID starts with “jstein”: `new LikeFilter( SearchableKeys.USER_ID, new StringValue("jstein*") );`
Principal name	`COMMON_NAME`	Searching principals with the name “Joshua Stein”: `new LikeFilter( SearchableKeys.COMMON_NAME, newStringValue("Joshua Stein") );`

Search criterion	Constant	Example
Language setting of users	`LANGUAGE`	Searching users with the language setting “English (US)”: `new EqualFilter( SearchableKeys.LANGUAGE, new LocaleValue("en_US" ) );`
Setting of the option *Trusted login* in the user settings	`TRUSTED_LOGIN`	Searching users who are allowed a trusted login: `new EqualFilter( SearchableKeys.TRUSTED_LOGIN, new BooleanValue( true ));`
Setting of the option *WCM access* in the user settings	`VIP_ACCESS`	Searching principals whose access to the WCM system has been deactivated: `new EqualFilter( SearchableKeys.VIP_ACCESS, new BooleanValue( false ));`
E-mail address	`MAIL`	see example below

Note: If you use a Livelink user administration, only the constants `USER_ID`, `VIP_ACCESS`, and `COMMON_NAME` can be used for searching principals. `COMMON_NAME` can only be used for searching the last name of users.

For detailed information on the individual constants, refer to the Javadoc on the `SearchableKeys` interface.

Examples

The following example searches for all principals whose e-mail address ends with the string “company.example”.

```
//login and get AdminHandler
ContextId cid = VipRuntime.getContextHandler().login(...);
AdminHandler ah = VipRuntime.getAdminHandler();

//construct filter expression for searching principals with the domain
//"company.example" in their e-mail address. A LikeFilter is used.
LikeFilter mailFilter = new LikeFilter(SearchableKeys.MAIL,
                                  new StringValue("*company.example"));

//search principals based on this filter
List principals = ah.getPrincipals(cid, AdminHandler.USER_PRINCIPALS,
                                          null, mailFilter, 0, -1);
```

The second example shows how to combine two filters. It searches for all principals with the language setting English and disabled WCM access.

```
//login and get AdminHandler
ContextId cid = VipRuntime.getContextHandler().login(...);
AdminHandler ah = VipRuntime.getAdminHandler();

//construct filter expression for searching principals with
//deactivated access to the WCM system
EqualFilter vipAccessFilter = new EqualFilter(SearchableKeys.VIP_ACCESS,
                                           new BooleanValue(false));
//construct filter expression for searching principals with
//language setting=English
//combine this filter with the vipAccessFilter constructed above
EqualFilter languageFilter = new EqualFilter(SearchableKeys.LANGUAGE,
                                            new LocaleValue("en_US"));
AndFilter filterCombination = new AndFilter(vipAccessFilter,
                                           languageFilter);
//perform searching, the principals found have the language setting
//"English" and their access to the WCM system is deactivated
List principals = ah.getPrincipals(cid, AdminHandler.USER_PRINCIPALS,
                                    null, filterCombination, 0, -1);
```

Deleting Principals

Principals are deleted by means of the `remove` method of the `AdminHandler` interface. The respective principal is removed from the user administration and from all assignments (to other principals, functional areas, and websites).

Notes:

- If the principal has already been assigned to WCM objects, the principal will not be removed automatically from the respective access control lists. This must be done via the WCM Java API or by means of the Content client.
- If you delete an LDAP principal in the WCM system, the corresponding entry will be removed completely from the LDAP directory service.
- If you delete a Livelink user, this user is marked as deleted in Livelink, but not removed from the database.

Setting WCM Attributes in LDAP

If an LDAP directory service is used for managing the user data of the WCM system, the principals stored in the LDAP server must have the WCM attributes. This gives them access to the WCM system.

> **Note:** For information on configuring the LDAP integration for Livelink WCM Server, refer to the Livelink WCM Server Installation Manual.

The required WCM attributes for principals in the LDAP directory service can also be set via the WCM Java API.

Use the `getUnassignedLDAPPrincipals` method to retrieve the principals within a certain search node that are not yet equipped with the WCM attributes. The returned list contains the “distinguished names” (DN) of the principals as `StringValues`.

These principals can then be equipped with the WCM attributes by means of the methods `importLDAPPrincipalAsUser`, `importLDAPPrincipalAsGroup`, and `importLDAPPrincipalAsRole`. In addition to the `ContextId`, you have to specify a `StringValue` with the DN of the principal to be imported as parameter.

Example

In the LDAP directory services, there is a node “o=company, c=en”. Below this node, there is, among other things, a node for users (“ou=users”) and a node for groups (“ou=groups”). These two nodes contain the principal entries that do not yet have the WCM attributes and that are to be equipped with these attributes via the WCM Java API.

The following figure illustrates the LDAP structure:

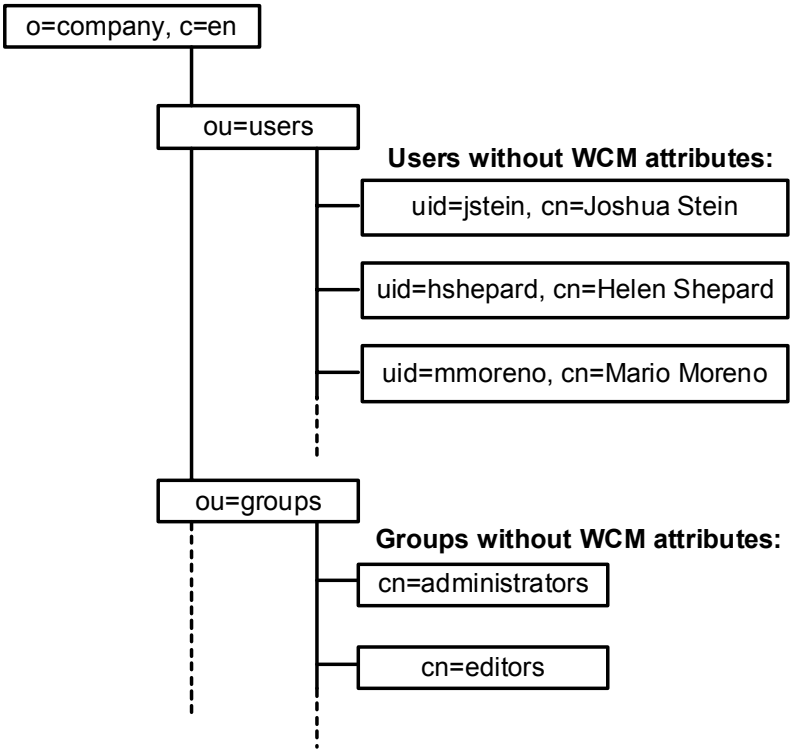


Fig. 7 – LDAP schema – example

The following example searches the nodes ou=users and ou=groups for principals that do not yet have the WCM attributes and assigns the attributes to the principals found.

```
//login and get AdminHandler
ContextId cid = VipRuntime.getContextHandler().login(...);
AdminHandler ah = VipRuntime.getAdminHandler();

//retrieve principals without WCM attributes in the LDAP context
//"o=company, c=en, ou=users"
//The returned list contains the LDAP DNs as StringValue objects
List unassignedUsers = ah.getUnassignedLDAPPrincipals(cid, "ou=users,
                                                    o=company,c=en", false);

//set WCM attributes for these users
Iterator usrIter = unassignedUsers.iterator();
while ( usrIter.hasNext() )
{
  StringValue usrDN = (StringValue) usrIter.next();
  ah.importLDAPPrincipalAsUser(cid, usrDN);
}

//retrieve principals without WCM attributes in the LDAP context
//"o=company, c=en, ou=groups" and assign the WCM attributes to
//these principals
List unassignedGroups = ah.getUnassignedLDAPPrincipals(cid, "ou=groups,
                                                    o=company,c=en", false);
Iterator grpIter = unassignedGroups.iterator();
while ( grpIter.hasNext() )
{
  StringValue grpDN = (StringValue) grpIter.next();
  ah.importLDAPPrincipalAsGroup(cid, grpDN);
}

...
```

Editing Rights of Principals

Livelink WCM Server differentiates two different types of permissions – administration rights and object rights. The administration rights determine which users are allowed to access the administration functions of Livelink WCM Server. The object rights determine which functions are available to the user when editing WCM objects.

The `AdminHandler` interface provides functions for assigning administration rights and default object rights to principals. In the WCM Java API, the access rights are represented by the following interfaces:

- `VipAdminPermission` (see the following section)
- `VipObjectPermission` (see “Default Object Rights of Principals” starting on page 69)

Administration Rights of Principals

Principals with administration rights have access to the configuration, user administration, and system administration as well as to the system overview of Livelink WCM Server. It is possible to assign graded access rights. Thus, a user may only be allowed to view and edit parts of the administration.

To perform specific functions of the WCM Java API, the user that is used for logging in an agent needs administration rights. For example, users can only be added if the administration right “Create, change, delete principal” has been assigned.

Certain rights can only be assigned in combination with other rights, i.e. they include other rights. Example: if you assign a principal the right “Create, modify, delete configuration entry”, this principal will automatically be given the rights “Access to configuration” and “Change configuration entry”.

The following table gives an overview of the individual rights and their dependencies. It also shows the constants defined for the rights in the `VipAdminPermission` interface. Individual rights, such as “Access to user administration”, “Access to configuration”, or “Access to system administration”, have a direct impact on the views displayed in the Admin client. The right “Website administration” influences individual functions both in the Admin client and in the Content client.

Table 3 – The individual administration rights

Right (Constant)	Possible actions
Read access to user administration (API) (USER_MANAGEMENT_READ_ACCESS)	If a server agent is to read the user information, the user used for logging in the agent must have this right.
Access to user administration (USER_MANAGEMENT_ACCESS)	Read access to the user information In the Admin client, the tree items *Users*, *Groups*, and *Roles* in the *User administration* view are only displayed if the user has this right.
Change principal (MODIFY_PRINCIPAL)	Change the settings of users, groups, and roles, as well as the assignments between these items, e.g. assignment of a user to a group. To be able to change the assignment of principals to functional areas and websites, the right “Change assignment to website/functional area” is also required. Includes the right “Access to user administration”
Change assignment to website/functional area (MODIFY_PRINCIPAL_ASSIGNMENT)	Change the assignments of users, groups, and roles to websites and functional areas Includes the right “Access to user administration”
Create, modify, delete principal (CREATE_MODIFY_REMOVE_PRINCIPAL)	Add, edit, and delete users, groups, and roles, as well as configure existing principals from an LDAP directory service as WCM principals Includes the rights “Change principal”, “Access to user administration”, “Change assignment to website/functional area”
Change administration rights (SET_ADMIN_ACL)	Edit the administration rights of principals Includes the right “Access to user administration”

Right (Constant)	Possible actions
Access to configuration (CONFIG_MANAGEMENT_ACCESS)	Read access to the configuration of Livelink WCM Server In the Admin client, the *Configuration* view is only displayed if the user has this right.
Change configuration entry (MODIFY_CONFIG_ENTRY)	Edit items in the *Configuration* view of the Admin client, including assignments between the items, e.g. assignment of proxy Content servers to websites Includes the right “Access to configuration”
Create, modify, delete configuration entry (CREATE_MODIFY_REMOVE_CONFIG_ENTRY)	Add, edit, and delete items in the *Configuration* view of the Admin client, e.g. websites or deployment systems Includes the rights “Change configuration entry” and “Access to configuration”
Access to system administration (SYSTEM_MANAGEMENT_ACCESS)	Read access to system administration and system overview In the Admin client, the *System administration* and *System overview* views are only displayed if the user has this right.
Change system status (CHANGE_SYSTEM_STATE)	Log out users, change run levels of servers and websites, abort actions in progress on a server, and use the commands in the *Tools* menu of the Admin client Includes the right “Access to system administration”

Right (Constant)	Possible actions
Website administration (WEBSITE_ADMINISTRATION)	Admin client: manage object types, attribute sets, and object categories; configure converters; change website run levels Content client: manage object types, attribute sets, and object categories For working in the views *Configuration* and *System administration* of the Admin client, the rights “Access to configuration” and “Access to system administration” must also be granted.

Use the `getAdminPermissions` method of the `User`, `Group`, or `Role` interface to read the administration rights of a principal. The returned list contains the constants of the `VipAdminPermission` interface representing the administration rights of the principal.

To edit the administration rights of a principal, use the `setAdminRights` method of the `AdminHandler` interface. The constants for the administration rights are supplied in a list.

A new user does not yet have any administration rights. The following example shows how to assign administration rights to a user:

```
//login and get AdminHandler
ContextId cid = VipRuntime.getContextHandler().login(...);
AdminHandler ah = VipRuntime.getAdminHandler();

//load profile of user 'Joshua Stein'
User usr = ah.getUserProfile(cid, "jstein",
                             AdminHandler.INDIRECT_ASSIGNMENTS);

//grant the administration rights MODIFY_PRINCIPAL and SET_ADMIN_ACL
//to the user
List admGranted = new LinkedList();
admGranted.add(VipAdminPermission.MODIFY_PRINCIPAL);
admGranted.add(VipAdminPermission.SET_ADMIN_ACL);
```

```
ah.setAdminPermissions(cid, usr, admGranted);
```

Default Object Rights of Principals

The default object rights are used as predefined settings for the object rights when a principal is added to the access control list of a WCM object. The exact access settings for a WCM object are part of the metadata of the object and can be changed via the `ObjectHandler` interface (see section “Editing Access Control Lists” on page 141).

The `VipObjectPermission` interface provides the following constants for the object rights:

Table 4 – Constants of the access rights for WCM objects

Access right	Constant
Read	`VipObjectPermission.READ`
Change object	`VipObjectPermission.WRITE`
Delete	`VipObjectPermission.DELETE`
Create	`VipObjectPermission.CREATE`
Release	`VipObjectPermission.RELEASE`
Change metadata	`VipObjectPermission.WRITE_META`
Change rights	`VipObjectPermission.CHANGE_RIGHTS`
Move and copy	`VipObjectPermission.TREE_OPERATIONS`
Read (Production)	`VipObjectPermission.READ_PRODUCTION`

Use the `getInitialPrincipalRights` method of the `User`, `Group`, or `Role` interface to read the default object rights of a principal. The method returns an object of the `InitialPrincipalRights` class. This class provides methods for requesting and setting granted and denied object rights.

The object rights that are set by means of the methods provided by `InitialPrincipalRights` must subsequently be saved as default object rights of the principal. Use the `setInitialRights` method of the `AdminHandler` interface for this purpose.

A new user does not yet have any default object rights. The following example illustrates how to set default object rights for a user:

```
//login and get AdminHandler
ContextId cid = VipRuntime.getContextHandler().login(...);
AdminHandler ah = VipRuntime.getAdminHandler();

//load profile of user 'Joshua Stein'
User usr = ah.getUserProfile(cid, "jstein",
                             AdminHandler.INDIRECT_ASSIGNMENTS);

//read the default object rights of the user
InitialPrincipalRights initialRights = usr.getInitialRights();

//The rights CREATE, DELETE, and READ are to be granted to the user
//These rights have to be created as a list
List granted = new LinkedList();
granted.add(VipObjectPermission.CREATE);
granted.add(VipObjectPermission.DELETE);
granted.add(VipObjectPermission.READ);

//The rights WRITE and TREE_OPERATIONS are to be denied to the user
List denied = new LinkedList();
denied.add(VipObjectPermission.WRITE);
denied.add(VipObjectPermission.TREE_OPERATIONS);

//Set objects right lists at the InitialRights object
initialRights.setDeniedPermissions(denied);
initialRights.setGrantedPermissions(granted);
```

```
//Save the changed object rights with the user by calling
//setInitialRights at the AdminHandler
ah.setInitialRights(cid, usr, initialRights);
```

Editing Functional Areas

In Livelink WCM Server, users must be directly or indirectly assigned to functional areas in order to perform certain actions via the clients, e.g. in the context of website management.

> **Note:** The functional areas of users are only checked when WCM objects are edited via the clients of Livelink WCM Server. They are not checked when objects are edited via the WCM Java API.

Functional areas perform two essential tasks:

- They determine which types of objects users are allowed to add, check out, and check in via the clients. Some functional areas, such as “Basic”, are by default assigned to object types. Only users that have the corresponding functional area can add, check in, and check out objects of this object type.

  Use the `getFunctionalArea` method of the `ObjectHandler` interface to retrieve the assignment of an object type to a functional area. The association of object types with functional areas can only be edited by means of the Admin client.

- They determine which views and dialog boxes are available to the user in the Content client. This way, you can specify exactly which actions the user is allowed to perform. For example, the Access rights dialog box is only displayed if the user has the functional area “Access rights dialog”.

In the WCM Java API, functional areas are represented by the `FunctionalArea` interface. The interface provides constants for the names of the predefined functional areas. If new functional areas are added via the Admin client or the WCM Java API, they are represented by their names.

Functional areas can be assigned to principals via the Admin client or the WCM Java API. This gives principals access to the respective functions for editing certain object types and to the respective functions of the Content client. For information on assigning functional areas to principals, refer to section “Editing Principal Assignments” starting on page 74.

The `AdminHandler` interface makes it possible to add, read, and delete functional areas. For more information, refer to the following sections:

- editing functional areas (see “Adding and Deleting Functional Areas” on page 72)
- reading functional areas (see “Retrieving Functional Areas of the WCM System” on page 73)

Adding and Deleting Functional Areas

Functional areas are added by means of the `FunctionalAreaFactory`. The factory is only used for creating functional area objects. To persistently add the new functional areas to the WCM system, the `add` method of the `AdminHandler` interface must be called afterwards.

The `FunctionalAreaFactory` interface has only one method: `createFunctionalArea`. To request an instance of the `FunctionalAreaFactory`, use the `getFunctionalAreaFactory` method of the `AdminHandler` interface.

When a functional area is added with the `createFunctionalArea` method, the name of the new functional area is supplied as string.

To delete functional areas, use the `remove` method of the `AdminHandler` interface.

Retrieving Functional Areas of the WCM System

To retrieve the functional areas used within a WCM system and their specific properties, you can use the methods `getFunctionalArea` and `getFunctionalAreas` of the `AdminHandler` interface.

The `getFunctionalAreas` method returns a list of `FunctionalArea` objects, yet without their properties, i.e. without users, groups, and roles. In this case, the `hasAssignment` method returns `false`. By means of the `getFunctionalArea` method, a completely loaded object can be obtained from the Administration server (`hasAssignment` returns `true`).

Example

The following code sample shows how to retrieve functional areas:

```
//login user, retrieve AdminHandler...
ContextId cid = VipRuntime.getContextHandler().login("..." , "...");
AdminHandler ah = VipRuntime.getAdminHandler();

//retrieve all available functional areas ..
List functionalAreas = ah.getFunctionalAreas(cid);
List faWithAssignments = new LinkedList();
while ( faIter.hasNext() )
{
FunctionalArea fa = (FunctionalArea)faIter.next();
   //fill a list with all functional areas including loaded assignments
faWithAssignments.add( ah.getFunctionalArea(cid, fa.getName(),
             AdminHandler.INDIRECT_ASSIGNMENTS));
}
....
```

Editing Principal Assignments

For the principals of the WCM system, various assignments can be made:

- assigning users to groups and roles

  This gives the users the rights of the respective groups or roles.

- assigning users as substitutes to other users

  After login, a substitute gets the object rights of the other user.

- assigning principals to websites

  If you assign a principal to a website, the respective users can access the objects of the website.

- assigning principals to functional areas

  The functional areas are used to determine which types of objects the users may edit via the clients of Livelink WCM Server and which dialog boxes and functions are available in the Content client.

For editing assignments, the `AdminHandler` interface provides the methods `defineAssignment` and `removeAssignment`.

Assigning principals to groups/roles, websites, and functional areas

The following example shows how to assign a user to a group. Afterwards, the user will be assigned to a website and all available functional areas.

```
//login and get AdminHandler
ContextId cid = VipRuntime.getContextHandler().login(...);
AdminHandler aH = VipRuntime.getAdminHandler();

//load user 'jstein' and group 'Administration'
//assign user 'jstein' to this group
User usr = aH.getUserProfile(cid, "jstein",
                             AdminHandler.INDIRECT_ASSIGNMENTS);
Group grpAdmin = aH.getGroupProfile(cid, "Administration");
aH.defineAssignment(cid, usr, grpAdmin);

//assign the user to website "InternetSite"
```

```
Website InternetSite = aH.getWebsite(cid, "InternetSite",
AdminHandler.NO_ASSIGNMENTS);
aH.defineAssignment(cid, InternetSite, usr);

//assign all available functional areas to the user
List faList = aH.getFunctionalAreas(cid);
Iterator faIter = faList.iterator();
while ( faIter.hasNext() )
{
FunctionalArea fa = (FunctionalArea)faIter.next();
ah.defineAssignment(cid, fa, usr);
}
```

Note: To determine which principals are assigned to a website, a functional area, or a group/role, use the respective `get` methods of the interfaces `Website` (e.g. `getUsers`), `FunctionalArea` (e.g. `getGroups`), `User` (e.g. `getRoles`), and `Group` or `Role` (`getMembers`).

Substitute assignments

If you edit the assignments between principals and supply the IDs of two users as arguments in the method `defineAssignment` or `removeAssignment`, the first user will be assigned/unassigned as the substitute of the second user.

```
//login and get AdminHandler
ContextId cid = VipRuntime.getContextHandler().login(...);
AdminHandler aH = VipRuntime.getAdminHandler();

//The user "mmoreno" is to be assigned as the substitute for "jstein"
User usrJst = aH.getUserProfile(cid, "jstein",
              AdminHandler.INDIRECT_ASSIGNMENTS);
User substitute = aH.getUserProfile(cid, "mmoreno",
                  AdminHandler.INDIRECT_ASSIGNMENTS);

//The first user specified in the parameters of the method
//defineAssignment becomes the substitute of the second user
aH.defineAssignment(cid, substitute, usrJst );
```

Saving User Settings

It is possible to save user-defined settings in the profile of a user. Settings that a user makes, for example, in the Configuration dialog box of the Content client (e.g. the representation of the toolbar, configured application classes) only apply to this user and are saved in the respective profile.

In the WCM Java API, the individual user settings are accessed via the `UserSettings` interface, which can be requested via the `User` interface. Internally, the user settings consist of key-value pairs. The keys are saved as `String`, the values are saved as `Value`. The `UserSettings` interface can be used to

- read values belonging to keys (method `UserSettings.getValue(String key)`)
- create and change values (method `UserSettings.setValue(String key, Value value)`)
- delete key-value pairs from the user settings (method `UserSettings.removeValue(String key)`)

The user settings can be grouped in order to represent different settings independently of each other. In addition to the Content client settings, it is, for example, possible to save an address for each user in the user settings. As the address data does not have anything to do with the Content client settings, it should be represented by a separate instance of a `UserSettings` object.

Theoretically, any number of different `UserSettings` instances can be saved for each user. For saving and differentiating the individual `UserSettings` instances, each instance has a name (of type `String`). This name can be retrieved by means of the method `UserSettings.getName`.

For retrieving the different user settings in the form of `UserSettings` objects and for creating new `UserSettings` instances, the `User` interface provides appropriate access methods.

Examples

In the first code sample, an address is to be saved in the settings of the user “jstein”.

```
ContextHandler ctxHandler  = VipRuntime.getContextHandler();

//Log in a user with Admin rights
ContextId cid = ctxHandler.login(...);
AdminHandler ah = VipRuntime.getAdminHandler();

//Load the profile of user 'jstein'
User usrJS = ah.getUserProfile( cid, "jstein",
                                AdminHandler.INDIRECT_ASSIGNMENTS );

//Create a UserSettings object for saving the address data
//'address' is used as the name for the settings
UserSettings adr = usrJS.createUserSettings("address");

//set address data
adr.setValue("Office/Street", new StringValue("Weidestr. 120"));
adr.setValue("Office/City", new StringValue("Hamburg"));
adr.setValue("Office/ZIPCode", new StringValue("22083"));

//save data permanently by calling update
//for the user 'jstein'
ah.update(cid, usrJS);
```

In the second code sample, all settings of the user “jstein” are read and output on the console.

```
//retrieve all UserSettings for the user. These are returned in a map
//containing the UserSettings objects as values and
//their names as keys
Map allUserSettings = usrJS.getAllUserSettings();

//The iterator contains the names of the UserSettings objects
//as strings.
Iterator usrSettingsIter = allUserSettings.keySet().iterator();
while ( usrSettingsIter.hasNext() )
{
      //Retrieve the UserSettings object
      UserSettings uS = (UserSettings)
                           allUserSettings.get(usrSettingsIter.next());

      System.out.println( "Settings '"+uS.getName()+"'
                                          contains keys/values : \n");

      //Iterate over the individual keys of the UserSettings and
      //output the associated values
      Iterator keyIter = uS.getEntryKeys();
      while ( keyIter.hasNext() )
      {
         String key = ( String ) keyIter.next();
         System.out.println( "\t key : "+ key + " , value : "+
                                                 uS.getValue(key));
      }
      System.out.println(" = = = = = = = = = = = = = = = = = = = = =\n");
}
```

CHAPTER 4

Event Processing

When statuses change in a WCM system, events are fired. These include *events* that are fired after the action (once the status change has occurred) as well as *preparation events* that are fired before the actual action.

In order to receive notification of the occurrence of certain events, server agents can register an *event listener* (`EventListener` or `DeploymentEventListener`) with the *event dispatcher* (`EventDispatcher` or `DeploymentEventDispatcher`). When an event occurs, the central event dispatcher distributes the event to all event listeners that have registered for it.

In general, each event has a definitive type, which represents the type of status change (or of the impending change). The possible event types are defined as constants in the `Event`, `DeploymentEvent`, `PrepareEvent`, and `DeploymentPrepareEvent` interfaces.

> **Note:** Events are fired on the generating server. Website-related and system-related events, as well as preparation events are triggered on the master Content server and should also be processed there. Deployment-related preparation events are available on the Content server on which the deployment system is installed.

Deployment events (`DeploymentEvents`) and deployment preparation events (`DeploymentPrepareEvents`) are not associated with a user context. This is an essential difference to website-related and system-related events (`Events`), as well as to preparation events (`PrepareEvents`).

An important difference between preparation events (`PrepareEvent` interface) and “normal” events (`Event` interface) is the fact that preparation events only have access to the user ID of the user who performed the relevant action. “Normal” events, however, retrieve an authentication object (`ContextId`) that allows them to perform additional actions under the logged-in user context. This means that

- “normal” events have all the rights of the user who performed the related action.
- preparation events cannot perform any changes to the actual action. Instead, a veto exception may be thrown before the action is executed. This exception prevents the action.

This chapter provides information on

- which “normal” events are defined in the interfaces `Event` and `DeploymentEvent` (see the following section).
- which preparation events are defined in the interfaces `PrepareEvent` and `DeploymentPrepareEvent` (see section “Preparation Events” starting on page 89).
- how to implement event listeners and event dispatchers (see sections “Event Listeners” on page 90 and “Event Dispatchers” on page 92).

Normal Events

In a WCM system, events are fired when the status of a WCM object changes, a page is being processed, or when the status of the entire system changes.

Accordingly, events belong to different categories:

- website-related events that refer to the status changes of a website's WCM objects
- system-related events that refer to the status change of the WCM system
- deployment-related events referring to status changes of pages, as well as to the creation or deletion of a deployment system

Website-related and system-related events are represented by objects of the `Event` interface (see the following section), deployment-related events are represented by the `DeploymentEvent` interface (see section "The `DeploymentEvent` Interface" on page 87).

The `Event` Interface

After a successful status change, events are fired, which are represented by objects of the `Event` interface.

Website events always refer to exactly one website. Use the `getConstraint` method to determine the name of this website (as a `String` object). For system-related events (e.g. an event indicating that a new website has been created), this method always returns `null`.

Each event has arguments that deliver more detailed information on the objective of the status change. For example, all website-related events have a reference to the related WCM object as an argument (OID). In general, you can determine the values of the arguments (the kind of which depend on the event type) by means of the `getArgument` method. All arguments can also be processed as an object of the `java.util.Map` container class.

Website-Related Events

The events of this category refer to the WCM objects of a website. For this event category, the `getConstraint` method of the `Event` interface provides the name of the website (as a `String` object). Each event contains the OID of the related WCM object as an argument. The constant `Event.ARG_OID` is used for this purpose. For the access, a corresponding cast is required, e.g.

```
ObjectId oid = (ObjectId)event.getArgument(ARG_OID);
```

The following events belong to this category:

- `OBJECT_ACL_CHANGED`
- `OBJECT_CHECKED_IN`
- `OBJECT_CHECKED_OUT`
- `OBJECT_CHECKOUT_UNDONE`
- `OBJECT_CREATED`
- `OBJECT_DELETED`
- `OBJECT_DESTROYED`
- `OBJECT_EXPIRED`
- `OBJECT_METADATA_CHANGED`
- `OBJECT_REJECTED_TO_EDIT`

- OBJECT_SUBMIT_AT_TO_PRODUCTION
- OBJECT_SUBMITTED_TO_PRODUCTION
- OBJECT_SUBMITTED_TO_QA
- OBJECT_CONTENT_SYNCHRONIZED
- OBJECT_WORKFLOW_ASSIGNED
- OBJECT_WORKFLOW_FORWARDED
- OBJECT_WORKFLOW_REMOVED
- TRANSACTION_END

For detailed information on the individual events and the associated arguments, refer to the Javadoc. For an example of a server agent which registers for the events `OBJECT_CREATED` and `OBJECT_METADATA_CHANGED`, refer to section “Converting Special Characters” on page 221.

System-Related Events

System-related events do not refer to the single WCM objects of a website, but rather to the WCM system as a whole. That is why there is no `Constraint` argument for these events that contains the name of the website. The `getConstraint` method from the `Event` interface always returns `null`.

The following events belong to this category:

- RUNLEVEL_DECREASES
- RUNLEVEL_INCREASES
- RUNLEVEL_IS
- USER_LOGIN
- USER_LOGOUT

- WEBSITE_DELETED
- WEBSITE_NEW

For detailed information on the individual events and the associated arguments, refer to the Javadoc.

Example

The following code sample shows a registration for the “Run level change complete” event (RUNLEVEL_IS). The agent is notified when the server has completed booting the configured website. For additional information on registration for events, refer to section “Event Dispatchers” on page 92.

```
public class DemoAgent
    implements de.gauss.vip.api.ServerAgent
{
   private final static de.gauss.vip.api.event.EventDispatcher
       eventDispatcher
           = de.gauss.vip.api.VipRuntime.getEventDispatcher();

    private final de.gauss.vip.api.event.EventListener eventListener
        = new DemoRunlevelListener();

    [...]
    private final String website;
    [...]
    public boolean start(String serverType, int majorVersion,
                         int minorVersion, String patchLevel)
    {
        [...]
        eventDispatcher.addListener(null,
                              de.gauss.vip.api.event.Event.RUNLEVEL_IS,
                              eventListener);
        return true;
    }

    public void stop()
    {
        eventDispatcher.removeListener(eventListener);
        [...]
    }
```

```
    private class DemoRunlevelListener
        implements de.gauss.vip.api.event.EventListener
    {
        public void performVipEvent(de.gauss.vip.api.event.Event e)
        {
          Integer level
=  (Integer)e.getArgument(de.gauss.vip.api.event.Event.ARG_NEW);
          String ws
=  (String)e.getArgument(de.gauss.vip.api.event.Event.ARG_WEBSITE);

          if (level.intValue()
                    == de.gauss.vip.api.admin.Runlevel.WEBSITE_UP
                && ws != null && ws.equals(website))
                System.out.println("Website " + website + " is up");
        }
    }
}
```

The `DeploymentEvent` Interface

Deployment events provide information on changes in the deployment. There are two types of events:

- events referring to the status of the deployment system (create and delete)
- events referring to the status of the pages of the deployment system (create, change, and delete)

  These events always refer to persistently saved generated pages (in the context of the dynamic deployment, there may also be pages that are temporarily generated). They result from a status change of the respective WCM object, which is transmitted by means of events, or from a `GENERATE_PAGE` event.

All events are fired after the respective change has been made. An important difference compared to website-related and system-related events is that there are no preparation events in the case of the deployment (the only exception being `PREPARE_FILE_CHANGED`). The reason for this is that the deployment only generates the representation of the WCM objects, but does not make any changes to the WCM objects. Conflicts that may occur are solved by the deployment itself. The generation of the pages is optimized in such away that in the case of changes the respective page is generated only once, if possible.

Deployment events are represented by objects of the `DeploymentEvent` interface and always refer to exactly one deployment system. The name of this deployment system (as a `String` object) can be retrieved by means of the method `getConstraint`.

Each event has arguments that deliver more detailed information on the objective of the status change. For example, all deployment-related events have a reference to the related WCM object as an argument (OID). In general, you can determine the values of the arguments (the kind of which depend on the event type) by means of the `getArgument` method. All arguments can also be processed as an object of the `java.util.Map` container class.

The following events belong to this category:

- `FILE_CHANGED`
- `FILE_DELETED`
- `FILE_URL_CHANGED`
- `DEPLOYMENT_SYSTEM_CREATED`
- `DEPLOYMENT_SYSTEM_DELETED`

For detailed information on the individual events and the associated arguments, refer to the Javadoc.

Preparation Events

The `PrepareEvent` Interface

Preparation events for website-related and system-related actions are represented by the `PrepareEvent` interface. They are fired before the underlying action is executed. This provides the possibility to review the impending status change and, if necessary, prevent it (by a `VetoException`). Each preparation event contains a reference to the related WCM object as an argument.

> **Note:** Preparation events for website-related and system-related actions are only available on the master Content server.

The following events belong to this category:

- `PREPARE_ASSIGN_WORKFLOW`
- `PREPARE_CHECKIN_OBJECT`
- `PREPARE_CHECKOUT_OBJECT`
- `PREPARE_CREATE_OBJECT`
- `PREPARE_DELETE_OBJECT`
- `PREPARE_DESTROY_OBJECT`
- `PREPARE_FORWARD_WORKFLOW`
- `PREPARE_OBJECT_CHANGE_METADATA`
- `PREPARE_REJECT_OBJECT_TO_EDIT`
- `PREPARE_RELEASE_OBJECT`
- `PREPARE_REMOVE_WORKFLOW`
- `PREPARE_SUBMIT_OBJECT_TO_QA`
- `PREPARE_UNDO_CHECKOUT_OBJECT`

For detailed information on the individual events and the associated arguments, refer to the Javadoc. For an example of a server agent that throws a `VetoException`, refer to section “Vetoing Metadata Changes” on page 212.

The `DeploymentPrepareEvent` Interface

Preparation events for deployment-related actions are represented by the `DeploymentPrepareEvent` interface. Before a persistently stored page is created or changed, the corresponding deployment system informs registered event listeners by means of the preparation event `PREPARE_FILE_CHANGED`. This provides the possibility to review the impending change and, if necessary, prevent it (by a `VetoException`).

Notes:

A veto against page generation usually results in inconsistencies, i.e. the page does not reflect the current state of the WCM object any longer. For this reason, you should not use this veto possibility.

Deployment-related preparation events are available on the Content server on which the deployment system is installed.

Event Listeners

To be able to react to certain events, an agent must register for the desired event by means of an event listener.

When the event occurs, the event dispatcher informs all event listeners registered for this event by calling their method `performVipEvent` or `performVipDeploymentEvent` with the respective event as an argument. The same applies to preparation events. In this case, the method `performVipPrepareEvent` or `performDeploymentPrepareEvent` is called with the preparation event as an argument.

When an event listener receives an event (e.g. `OBJECT_METADATA_CHANGED`), it may be possible that the respective WCM object does no longer exist in the current version, e.g. because it has been finally deleted in the meantime (`destroy` action). Unlike preparation events, events are sent asynchronously. For example, a staging action, such as submit, may have been executed before the event for the previous step in the staging is fired. When the WCM object is read as the result of an event, the version of the object should be compared with the version of the event. This can be done in the method `performVipEvent`. If the versions are different or if the object cannot be found, you can try to read the archived object (`ObjectHandler.get(oid,version)`). The version of a WCM object is saved in the argument `ARG_VERSION` for each event.

Event listeners are provided by the following interfaces:

- `EventListener` for website-related and system-related events

  Event listeners for website-related and system-related events must implement the interface `EventListener` (`de.gauss.vip.api.event`). This interface contains a method that is called when a registered event occurs. This method is called when the action causing the event has been completed.

- `DeploymentEventListener` for deployment-related events

  Event listeners for deployment events must implement the interface `DeploymentEventListener` (`de.gauss.vip.api.event`). This interface contains a method that is called when a registered event occurs. This method is called when the action causing the event has been completed.

- `PrepareEventListener` for website-related and system-related preparation events

This listener is required for processing website-related and system-related preparation events. There is a preparation event for almost every website-related and system-related event (see section “The `PrepareEvent` Interface” starting on page 89). In this listener, it is possible to veto performing an action. To do this, a `VetoException` (`de.gauss.vip.api.event`) must be thrown.

The `UserMessage` class is used for the message argument in the `VetoException`. The message is displayed in the client when the exception is thrown.

- `DeploymentPrepareEventListener` for deployment-related preparation events

  This listener is required for processing deployment-related preparation events. There is one deployment-related preparation event (see section “The `DeploymentPrepareEvent` Interface” starting on page 90). In this listener, it is possible to veto generating a page. To do this, a `VetoException` (`de.gauss.vip.api.event`) must be thrown.

Event Dispatchers

When an event occurs, central event dispatchers distribute the event to all event listeners that have registered for it. For this purpose, the event listeners must register with the event dispatcher.

Two event dispatchers are available:

- `EventDispatcher` for distributing website-related and system-related events

This interface can be retrieved via `VipRuntime.getEventDispatcher`. It offers methods for registering for website-related and system-related events. In the case of website-related events, the name of a website must be supplied for the `constraint` argument of the `addListener` or `removeListener` method.

To register a listener, an event type (of the `Event` or `PrepareEvent` interface) is specified. It is possible to use the same listener for several event types and to separate them in the corresponding methods. However, the best solution is to create a separate class or object for each event type.

- `DeploymentEventDispatcher` for distributing deployment-related events

  This interface can be retrieved via `VipRuntime.getDeploymentEventDispatcher`.

  The interface provides methods that can be used to register for events relating to the deployment. In the `constraint` argument of the method `addListener` or `removeListener`, the name of the deployment system is supplied. To register a listener, an event type (of the `DeploymentEvent` or `DeploymentPrepareEvent` interface) is specified.

CHAPTER 5

Object Management

The WCM Java API provides various interfaces for editing WCM objects. The following diagram shows the object management components.

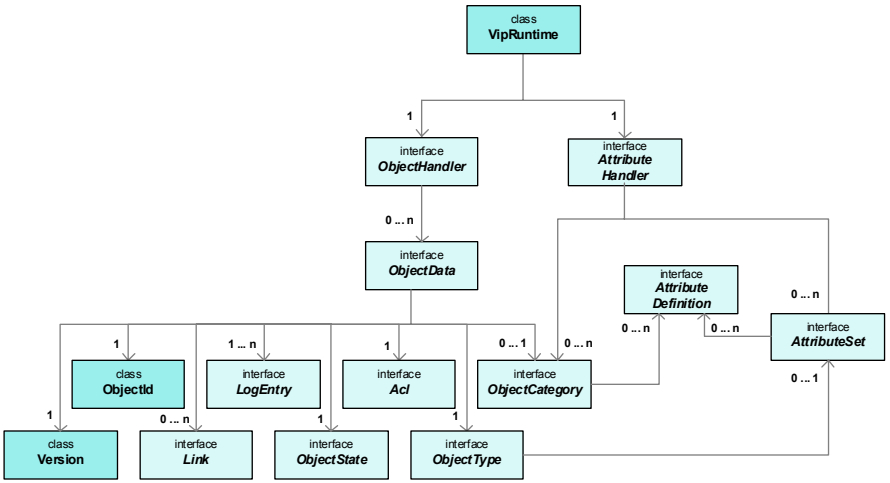


Fig. 8 – Object management components

The following interfaces are of central importance:

- `ObjectHandler` interface: This interface contains methods for adding and deleting WCM objects, for performing staging actions (e.g. submit or release), and for getting object data to be processed.

  See section “Actions of the `ObjectHandler` Interface” starting on page 108

- `ObjectData` interface: This interface contains methods for editing the metadata of WCM objects.

  See section “Editing Object Data – The `ObjectData` Interface” starting on page 121

- `AttributeHandler` interface: This interface provides methods for editing attribute sets and object categories.

  See section “Editing Special Attributes” starting on page 128

- `ACL` interface: This interface contains methods for editing the access control lists of WCM objects.

  See section “Editing Access Control Lists” on page 141

The `ObjectHandlerUtil` class provides auxiliary methods to facilitate access to frequently used objects of the WCM Java API (see section “The `ObjectHandlerUtil` Utility Class” on page 120).

Moreover, this chapter provides information on searching objects by means of filter functions (see section “Searching WCM Objects” starting on page 133).

Conditions for Editing Objects

Whether a WCM object can be edited and which functions are available for this purpose depends on several factors:

- the object status (see the following section)
- the data storage view on the current server (see “Data Storage Views” on page 100)
- the object's position in the assigned workflow (see “WCM Objects in the Workflow” on page 102)

- the type of the object (see “Object Types” on page 105)
- the access rights of the logged-in principal (see “Access Rights for WCM Objects” on page 107)

In certain cases, preconditions for child or parent objects must also be met. For example, an object can only be released if the parent topic has already been released.

Object Statuses – Constants and Possible Actions

Each WCM object has a certain status that describes the current processing state. The following statuses exist:

Table 5 – Object state constants

Status	Constant	Value
changed	`ObjectState.EDITED`	`edited`
checked out	`ObjectState.CHECKED_OUT`	`checked_out`
deleted	`ObjectState.DELETED`	`deleted`
rejected	`ObjectState.REJECTED`	`rejected`
released	`ObjectState.RELEASED`	`released`
delayed release	`ObjectState.PENDING_RELEASE`	`release_at`
submitted	`ObjectState.SUBMITTED`	`submitted`

In the WCM Java API, the status of a WCM object is represented by the `ObjectState` interface. In addition to the name of the status (see the constants above), you can check whether certain status transitions are possible. Moreover, a localized description of the status can be determined.

The `ObjectState` object belonging to a name is determined using a method of the `Website` object (`Website.getObjectState(name)`, see also “Information on Websites” on page 162 for more information.

Depending on the object status, different actions, such as submit or release, are possible for an object. Normally, a successfully executed action will change the object status of the respective WCM object. A WCM object submitted to Quality Assurance, for example, has the status `SUBMITTED` before it is released. If the WCM object is successfully released by the `release` method, it acquires the status `RELEASED`.

The following table offers an overview of the possible actions for the object statuses and the resulting status changes.

Table 6 – Object statuses and possible actions

Original object status	Possible actions	Resulting object status
`EDITED`	`change`	unchanged
	`checkOut`	→ `CHECKED_OUT`
	`copy`	Source object: unchanged Target object: → `EDITED`
	`delete`	→ `DELETED`
	`directRelease`	→ `RELEASED` → `PENDING_RELEASE`
	`move`	unchanged
	`restoreVersion`	unchanged
	`submit`	→ `SUBMITTED`
`CHECKED_OUT`	`checkIn`	→ `EDITED`
	`undoCheckOut`	back to the status the object had before

Original object status	Possible actions	Resulting object status
DELETED	destroy	Object is finally deleted from the WCM system
	reject	→ REJECTED
REJECTED	change	→ EDITED
	checkOut	→ CHECKED_OUT
	copy	Source object: unchanged Target object: → EDITED
	delete	→ DELETED
	move	→ EDITED
	restoreVersion	→ EDITED
	submit	→ SUBMITTED
RELEASED	change	→ EDITED
	checkOut	→ CHECKED_OUT
	copy	Source object: unchanged Target object: → EDITED
	delete	→ DELETED
	move	→ EDITED
	restoreVersion	→ EDITED

Original object status	Possible actions	Resulting object status
PENDING_RELEASE	change	→ EDITED
	checkOut	→ CHECKED_OUT
	copy	Source object: unchanged Target object: → EDITED
	delete	→ DELETED
PENDING_RELEASE	restoreVersion	Edit view: → EDITED QA view: unchanged
SUBMITTED	reject	→ REJECTED
	release	→ RELEASED → PENDING_RELEASE

Data Storage Views

According to the staging of Livelink WCM Server, there are three views of the data storage in a WCM system: Edit, QA, and Production. Depending on the object status, the WCM objects of a website are available in different views and can be edited. A new object, for example, is only available in the Edit view. It cannot be seen in the QA or Production view.

During staging, a WCM object goes through various object statuses and different views. The availability of the WCM object in the different data storage views can change as the result of an action. After release, the current state of a WCM object is also available in the Production view, for example. Further actions on the WCM object in the QA view, however, would not be possible until the object has been resubmitted to Quality Assurance.

The following figure illustrates the staging for processing WCM objects:

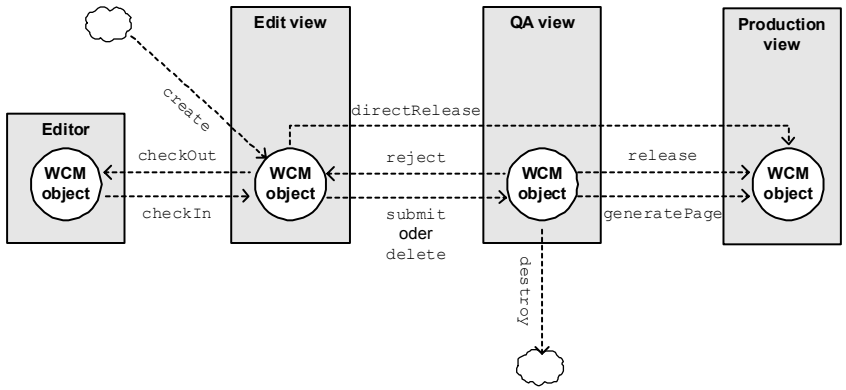


Fig. 9 – Staging for processing a WCM object

Which view of the WCM objects of a website is available on a server depends on the routing for the website. A master Content server always has all views. The views on a proxy Content server depend on which view this server gets in the routing chain between the Content servers. This is determined when the website is created or when proxy Content servers are assigned to a website.

It is, for example, recommendable that the proxy Content server used for editing the WCM objects gets the Edit view of the WCM objects. This view also comprises the QA and Production views and contains the largest amount of data. The proxy Content server used for the published website, by contrast, only needs the Production view of the WCM objects.

When accessing WCM objects, the data storage view used must be specified. For this purpose, the type constants in the `Website` interface are used. The following table shows which object statuses are available in the different views.

Table 7 – Visible statuses of WCM objects in the data storage views

View	Constant for data storage view `Website` interface	Status of the WCM objects
Edit	`DATA_VIEW_EDIT`	all
QA	`DATA_VIEW_QA`	all except `EDITED` and `CHECKED_OUT`
Production	`DATA_VIEW_PROD`	`RELEASED`

Older versions of the WCM objects are archived in a version table independent of the respective view.

> **Note:** The available views of the deployment systems of the WCM objects are determined by analogous type constants in the `DeploymentSystem` interface (see section “Information on the Deployment” starting on page 164).

The object management actions and the changes resulting from an action are only available in certain data storage views. In the QA view, it is, for example, not possible to add a WCM object, to submit a changed WCM object, etc. For this reason, the Javadoc on the object management actions always contains information about the data storage view in which the action is possible.

WCM Objects in the Workflow

In Livelink WCM Server, the predefined stages (editing, quality assurance, publication for productive operation) can be extended by user-defined workflow steps in order to realize editing by several editors or a multi-stage quality assurance process.

For this purpose, the object type “Workflow” exists. For adding objects of the type “Workflow”, the user requires the functional area “Workflow”. An object of type “Workflow” represents a so-called *workflow definition*, which contains the individual workflow steps. Workflow definitions are created and edited with the *Content Workflow Modeler* (workflow modeler for short). In the Content client, workflow objects can be assigned to WCM objects.

The workflow definitions created with the workflow modeler are saved in the XML Process Definition Language (XPDL). The format XPDL is XML-based and is the standard defined by the Workflow Management Coalition (WFMC) for interchanging process definitions.

A workflow definition has at least one *starting point* and at least one *end point*. When an WCM object is forwarded in the workflow and an end point is reached, the assignment between WCM object and workflow object is automatically removed. The following types of end points exist:

- `Release`: When this end point is reached, the WCM object is released. If a delayed release date is set for the object, the object gets the status “delayed release”.
- `Stop`: When this end point is reached, the assignment between workflow object and WCM object is removed. The status of the WCM object does not change.
- `Destroy`: When this end point is reached, the WCM object is irretrievably deleted.

A workflow definition may contain any number of *activities* and *transitions*. Each activity is assigned to a principal (user, group, or role) who may edit the object and forward it in the workflow. The three types of activities “Edit”, “QA”, and “Delete” represent the task of the assigned principal.

By means of the transitions, the WCM object is forwarded from one activity to the next. The necessary staging transitions are automatically performed in the background.

Note: It is not possible to perform normal staging transitions on WCM objects to which a workflow is assigned. The following actions are rejected with an error: `submit`, `release`, `toQa`, `reject`, `direct release`, `delete`, `destroy`.

The following staging transitions can be represented in the workflow. The specified actions are automatically performed in the background:

Table 8 – Workflow transitions and required staging actions

to from	Edit	QA	Delete	Release	Destroy
Edit	No action required	`submit`	`delete`	`direct release`[a]	`direct destroy`[b]
QA	`reject`	No action required	Transition impos-sible	`release`	Transition impossible
Delete	`reject`	Transition impossible	No action required	Transition impos-sible	`destroy`

(a) The action `direct release` releases the object directly. The functional area “Direct release” and the metadata item “Direct release” are not checked.

(b) The action `direct destroy` destroys the object, i.e. permanently deletes it from the WCM system. If necessary, the object is automatically submitted to Quality Assurance for deletion.

When a workflow is assigned to a WCM object, the latest released version is always used. Thus, the respective workflow object must be released first before the changed workflow definition can be used. The changed workflow definition does not affect WCM objects to which the workflow has already been assigned. These objects pass through the old version of the workflow.

For further information on workflows, refer to section “Performing Workflow Actions” starting on page 142.

Object Types

Each WCM object has an object type, which, on the one hand, classifies the content used (e.g. “HTML page” or “JSP page”) and, on the other hand, determines additional properties (e.g. “Template” or “Topic”). Thus, the object type defines the handling of the object within the WCM system (e.g. regarding page generation).

The `ObjectType` interface defines the object type of a WCM object. In addition to the unique identifier of an object type, further properties are specified, e.g. whether a content file is mandatory when a WCM object of this type is added.

The object type name can be assigned to the respective Java object using a method of the `Website` interface. The following example returns an object of the `ObjectType` interface on the basis of the object type name (`HTML`).

```
ObjectHandler oh = VipRuntime.getObjectHandler("demosite");
ObjectType ot = oh.getWebsite().getObjectType(ObjectType.HTML);
```

The `getDescription` method returns a localized description of the object type (e.g. for a client).

Every object type can be assigned an attribute set that defines special attributes for this object type. The `getAttributeSetName` method returns the name of the attribute set. Attribute sets are explained in detail in section “Editing Special Attributes” starting on page 128.

The actions that can be applied to a WCM object partly also depend on the type of the object. Changing the object type is only possible for some of the default object types.

Notes:

You can only delete an object type if no **current** versions of objects exist in the Edit, QA, or Production view that use this object type. Older versions of the objects with this object type may, however, exist.

If an object version is restored that uses the deleted object type, the object type will also be restored. The utility *Database clean-up* finally removes deleted object types if there are no older object versions that use these types.

Example

The following code sample shows how to output the object type and attribute set for an object with the OID 4711 and catch an `AccessDeniedException`.

```
//log in user, retrieve AdminHandler...
ContextId cid = VipRuntime.getContextHandler().login("..." , "...");

//retrieve ObjectHandler for website 'InternetSite'
ObjectHandler oh = VipRuntime.getObjectHandler("InternetSite");

//retrieve object with oid 4711 ...
ObjectData od =null;
try
{
od = oh.get(cid, new ObjectId("4711"));
}
catch ( AccessDeniedException aex )
{
System.out.println( "cannot access object with oid 4711 ! ");
aex.printStackTrace();
}

//retrieve object type
ObjectType type = od.getObjectType();
System.out.println( "object 4711 has type : "+
type.getDescription(Locale.getDefault()));

//retrieve the name of the attribute set belonging to the object type
String attrSetName =  type.getAttributeSetName();
```

```
AttributeSet attributes =
VipRuntime.getAttributeHandler("InternetSite").getAttributeSet(
      attrSetName);
System.out.println("attributes" + attributes);
....
```

Access Rights for WCM Objects

In Livelink WCM Server, there are nine access rights for WCM objects. Certain actions on WCM objects are only possible if the respective right for the current user exists directly or indirectly in the WCM object's access control list. Directly assigned rights are the rights of the individual user. Indirectly assigned rights are the rights of the group or role the user belongs to.

The object access rights are represented by constants of the `VipObjectPermission` interface (see table "Constants of the access rights for WCM objects" on page 69). For information on editing the access control list of a WCM object, refer to section "Editing Access Control Lists" on page 141.

Whether an action is permitted for a WCM object is checked as follows: The WCM system reads the WCM object's access control list (ACL) and checks whether the logged-in user has the right required for the action. The rights required for specific actions are listed in the Javadoc.

Actions of the `ObjectHandler` Interface

The `ObjectHandler` interface provides access to the WCM objects of a website. To request the `ObjectHandler` via the method `VipRuntime.getObjectHandler`, the name of the website must be supplied as an argument. The names of all websites managed by the agent's server can be retrieved by means of `VipRuntime.getCurrentServer().getWebsiteNames`. The respective website contains, for example, the object types and statuses used. The `Website` object can be retrieved by means of `ObjectHandler.getWebsite`.

The `ObjectHandler` interface provides different methods for performing actions on WCM objects. The following section gives you an overview of these methods.

The methods of the `ObjectHandler` interface are divided into synchronous and asynchronous methods. For an overview, refer to section "Synchronous and Asynchronous Methods" on page 111.

Several actions on WCM objects can only be performed within one transaction, see "Transactions" on page 113.

Every action is executed within a context. A distinction is made between a user context and a transaction context, see "User and Transaction Contexts" on page 117.

You also have to consider that actions on objects and deployment are independent of each other. When an action has been finished, this does not necessarily mean that the deployment systems have already generated pages that reflect the changes made to the WCM objects. Please also read section "Actions and Deployment" on page 118.

Overview of the `ObjectHandler` Methods

The methods of the `ObjectHandler` interface can be used for the following tasks:

Table 9 – Main actions of the `ObjectHandler` interface

General task	Individual task	Methods
Adding objects	Add a new object	`create`
	Import objects	`multiImport`
Staging actions	Check out	`checkOut`
	Undo check-out	`undoCheckOut`
	Check in	`checkIn`
	Submit	`submit`
	Release	`release` `directRelease`
	Reject	`reject`
Retrieving and editing objects	Get objects	`get`
	Edit metadata	`change`
	Add remark	`addRemark`
Copying and moving	Copy	`copy`
	Move	`move`
Converting objects		`convertContent`
Searching objects		`filter`
Restoring old version		`restoreVersion`

General task	Individual task	Methods
Page generation (deployment)	Delete a generated page from the file system	`depublishPage`
	Regenerate a page for an individual WCM object	`generatePage`
	Regenerate a page for a topic and its subordinate objects	`generateTopic`
Deleting objects	Delete objects	`delete`
	Destroy object, i.e. delete it permanently from the WCM system	`destroy`

The Javadoc contains detailed information on the individual methods. It contains information on the required access rights, the possible previous object statuses, and the resulting status. For each method, it is specified in which view the respective action is available (i.e. can be performed).

Note: Since all data storage views are available on the master Content server, all actions are permitted and the resulting object statuses are visible on this server.

Example

The following code sample illustrates the methods for adding, changing, submitting, releasing, deleting, and destroying an object:

```
//log in user...
ContextId cid = VipRuntime.getContextHandler().login("..." , "...");

//retrieve ObjectHandler for website 'InternetSite'
ObjectHandler oh = VipRuntime.getObjectHandler("InternetSite");
```

```
//create an initial ObjectData object; set a subtitle ..
ObjectData odNew = oh.createInitialObjectData(new ObjectId("1"), null,
                                                    html, "test object");
odNew.setSubtitle("testing object creation..");

//create a new object from the initial object data
ObjectId oidNew = oh.create(cid, odNew, null, "create object");

//submit, release, delete, and destroy this object...
List oids = new LinkedList();
oids.add( oidNew);

oh.submit(cid, oids , null, "submit to qa..", null );
oh.release(cid, oids, "release object to production", null);
oh.delete( cid, oids, "delete object", null);
oh.destroy(cid, oids, " destroy object permanently ", null);
```

Synchronous and Asynchronous Methods

The methods of the `ObjectHandler` interface are divided into synchronous and asynchronous methods. Asynchronous methods can be recognized by the suffix `-Async`. All other methods are synchronous methods.

The following table contains the methods of the `ObjectHandler` interface that can be called synchronously or asynchronously.

Table 10 – Synchronous and Asynchronous Methods

Synchronous methods	Asynchronous methods
change	changeAsync
copy	copyAsync
delete	deleteAsync
depublishPage	depublishPageAsync
destroy	destroyAsync
directRelease	directReleaseAsync

Synchronous methods	Asynchronous methods
generatePage	generatePageAsync
move	moveAsync
multiImport	multiImportAsync
reject	rejectAsync
release	releaseAsync
submit	submitAsync
undoCheckOut	undoCheckOutAsync

The difference between synchronous and asynchronous methods lies in the time of their return. If a synchronous method is called, it returns when the actual task has been completed and the local deployment has been informed (the deployment, however, has not yet been executed at this time). In contrast, the asynchronous method returns almost immediately after its call. The next WCM Java API method can thus already be called even though the actions required for executing the first method are still running in the background.

For clarification, we assume that you want to copy a relatively large subtree to another topic. After that, you want to execute various other actions in another section of the website. After a synchronous call of the `copy` method, the following methods will not be executed until the server has finished copying all objects contained in the subtree (except for deployment). If the corresponding asynchronous method `copyAsync` is called, however, the respective copying actions will be started on the server, but the method returns immediately and the subsequent methods are executed.

The use of asynchronous methods is particularly useful when you trigger actions that may take some time, but do not want to wait until these

actions are concluded. Since the asynchronous method returns immediately after the action has been started on the server, you can perform other actions on *unaffected objects* in the meantime. If an error occurs during an asynchronous action, you can undo the entire action. In case of synchronous methods, this is not necessarily the case.

Transactions

Several actions on WCM objects can be performed within a transaction. This means that the changes do not come into effect until the transaction has been explicitly completed.

Important notes

- To ensure the consistency of the WCM system, the methods `commitTransaction` and `rollbackTransaction` must be executed as early as possible. We recommend that you insert Commit and Rollback mechanisms for started transactions in a `finally` block. This ensures that no unfinished transactions are left behind in case of errors.

  Code sample

```
public class TransactionExample
{
   public void mySpecialOperation(ContextId cid,ObjectHandler oh,
      ObjectId firstObject, ObjectId secondObject) throws
      VipApiException
   {
      boolean commitTransaction=false;
      ContextId transactionId = null;
      try
      {
         transactionId = oh.beginTransaction
            (cid,"special operation",2);
         // do some actions
         oh.addRemark(transactionId, firstObject, "first remark");
         oh.addRemark(transactionId, secondObject, "second remark");
         // all actions done
         commitTransaction=true;
```

```
        }
        finally
        {
           if (transactionId != null)
           {
              if (commitTransaction)
                 oh.commitTransaction(transactionId);
              else oh.rollbackTransaction(transactionId);
           }
        }
     }
}
```

- It is strictly forbidden to request a transaction while waiting for results referring to this transaction. **Do not use transactions asynchronously!** Transactions are a limited programming concept that must be used carefully as they may prevent other users from working in the WCM system.

Starting and Ending Transactions

A transaction is requested using the `beginTransaction` method of the `ObjectHandler` interface. A transaction context (an object of the `ContextId` interface) is returned and can be used as an argument for additional actions (e.g. `create`, `submit`, or `release`). All actions using this transaction context are executed within the transaction and do not come into effect until the `commitTransaction` method is called. The changes are not carried out if `rollbackTransaction` is called.

By means of the `beginTransaction` function, you can combine several method calls to form a single action. With `rollbackTransaction`, the running transaction is aborted or, rather, the executed changes are undone. When the transaction has been performed successfully, corresponding changes are confirmed by the `commitTransaction` method and thus made permanent. This means that all actions combined in a transaction are either executed or aborted as a whole.

After a transaction has been confirmed with `commitTransaction`, the event `TRANSACTION_END` is fired. For detailed information on event processing, refer to chapter 4 “Event Processing”.

Notes:

Transactions can only be requested and completed on the master Content server, i.e. the `beginTransaction` and `commitTransaction` methods are available only there.

If actions are executed within a transaction, the respective events will not be triggered until the transaction has been successfully completed.

Watching the Progress of Transactions

The progress of the transaction processing can be determined using the `Progress` object. This progress control can be updated by another API request (`Progress.update`).

A `Progress` object can have the following states:

- `RUNNING`: The transaction is currently in progress.
- `FINISHED`: The transaction was finished successfully.
- `DEPLOY`: A locally installed deployment system still has jobs for this transaction.
- `DEPLOY_FINISHED`: The deployment jobs for this transaction (on the locally installed deployment systems) have been completed.
- `ERROR`: The transaction was cancelled due to an error or an explicit abort request (`rollbackTransaction`).
- `INVALID`: An attempt was made to determine the progress of an already completed transaction. The transaction context is invalid.

If the current server has its own deployment systems, the state of the `Progress` object changes from `FINISHED` to `DEPLOY`. In this case, you can wait until all locally installed deployment systems have processed the deployment jobs belonging to the transaction (all pages generated). You can also retrieve the current number of jobs and the number of jobs processed so far.

The progress of a transaction may look as follows:

- `INVALID`
- `RUNNING` → `ERROR`
- `RUNNING` → `FINISHED`
- `RUNNING` → `FINISHED` → `DEPLOY` → `DEPLOY_FINISHED`

Notes:

The progress can only be determined as long as the transaction context remains valid. At the end of a transaction, the transaction context is invalid (whether or not the transaction was successful).

The progress of the deployment can only be determined for locally installed deployment systems.

Example

The following WCM objects are to be synchronously submitted and asynchronously released (OIDs): 1, 3, 10, 15 (execution on the master Content server). The progress of the release action is displayed. After that, the sample program waits until all locally installed deployment systems have finished the jobs belonging to the transaction.

```
ObjectHandler oh = VipRuntime.getObjectHandler("website");
ContextId cid = VipRuntime.getContextHandler().login(userid,password);
List l = ObjectHandlerUtil.toOIDList("1,3,10,15");
oh.submit(cid,l,null,null,null);
ContextId tid = oh.releaseAsync(cid,l,null,null);
Progress prog = oh.getProgess(tid);
```

```
while (prog.getState()==Progress.RUNNING)
{
   System.out.println("state="+prog.getMessage(Locale.getDefault())+
      " step="+prog.getCurrentStep()+" max="+prog.getNumberOfSteps());
   prog.update();
   Thread.currentThread().sleep(200);
}
if (prog.getState()==Progress.ERROR)
{
   System.out.println("error ="+prog.getException().getMessage());
}
else // FINISHED
{
   prog.update();
   while (prog.getState()==Progress.DEPLOY)
   {
         Thread.currentThread().sleep(200);
         System.out.println("Local deployment active...");
         prog.update();
   }
}
```

User and Transaction Contexts

Every action is executed within a context. A distinction is made between a user context and a transaction context. Both contexts are represented by objects of the `ContextId` interface.

A user context is the result of the login process. It contains information on the logged-in user, in particular the user ID and information on the last action the user performed. A user context becomes invalid if no action has been performed for a certain period of time (see also section “Authentication and Contexts” starting on page 36).

A transaction context extends the user context. It contains a reference to the corresponding transaction. During a transaction, the progress of the action can be checked and the transaction may be aborted, if necessary. The transaction context becomes invalid as soon as the action has been completed. If it was successful, the system state will be updated afterwards.

Notes:

When a synchronous or asynchronous action has been finished, i.e. the transaction has been completed successfully, the object data on all servers is updated with the current data of the master Content server. However, the action does not wait for the deployment systems of the servers. For this reason, the URL of a deployment system might reference a resource that has not yet been generated by the deployment system.

The WCM system blocks objects that are currently being edited in a transaction. Thus, these objects are not available for actions running in parallel.

Actions and Deployment

The actual execution of an action continues until the corresponding changes in the WCM objects have been persistently saved on the master Content server and all proxy Content servers have been informed of the changes. When an action has been finished, this does not necessarily mean that the deployment systems have already generated pages that reflect the changes made to the WCM objects. Action and deployment are independent of each other. This must be taken into consideration when accessing the generated pages (after an action). The relevant deployment jobs may not yet have been carried out so that no URL is yet available for a page of a certain deployment system.

Use the following methods to retrieve information on the progress of the deployment:

- `Progress.getProcessedJobs` – Returns the number of completed deployment jobs (for a certain transaction).
- `Progress.getRemainingJobs` – Returns the number of deployment jobs (for a certain transaction) that have not yet been finished.

It is also possible to wait for the conclusion of deployment jobs (for a certain deployment system) connected with an action.

For this purpose, the additional parameter `DeploymentWaitInfo` can be used for the corresponding methods of the `ObjectHandler` interface when performing synchronous staging actions. This parameter can also be used to determine how long the system should wait for the deployment to finish. The following example adds a WCM object and waits a maximum of 10 seconds for the deployment system specified:

```
ObjectHandler oh = ...
ObjectData objectData = oh.createInitialObjectData(parentOID,
templateOID,
   type, title);
DeploymentWaitInfo dwi = new DeploymentWaitInfo("demowebsite_edit",
   10000);
ObjectId oid = oh.create(cid, objectData, content, remark, dwi);
if (!dwi.isTimeout())
{
    ObjectData od = oh.get(cid, oid);
    URL url = od.getURL("demowebsite_edit");
    ...
}
```

In the case of asynchronous staging actions, it is also possible to wait for the conclusion of the deployment. The `waitForDeployment` method of the `Progress` interface is used for this purpose. In the following example, the agent carries out an asynchronous import and waits for the specified deployment system for max. 10 seconds.

```
ContextId tid = oh.multiImportAsync(cid, parentOid, title,
   startFile, initialData, remark);
Progress prog = oh.getProgress(tid);
if (prog.waitForDeployment("demowebsite_edit", 10000))
{
   ... // deployment finished
}
```

Note: In order to wait for the conclusion of relevant deployment jobs, a bi-directional connection with the corresponding deployment system is required: in this case, communication with the deployment system must not be blocked by a firewall (in one direction).

The `ObjectHandlerUtil` Utility Class

The `ObjectHandlerUtil` class provides auxiliary methods to simplify access to frequently used API objects. Using the auxiliary methods of this class facilitates the following tasks:

- determining the object types of a website: Various methods of the `ObjectHandler` interface require an object type as an argument. In the WCM system, the object types are defined separately for each website and are thus retrieved via the `Website` interface. Accessing this information by means of the methods of the `ObjectHandler` or the `Website` interface is not very easy (see example).
- determining attribute sets and object categories
- creating OID lists: Most methods of the `ObjectHandler` interface require an OID list as an argument. The `ObjectHandlerUtil.toOIDList` method can be used to create a list of one or more OIDs from a string.
- determining the product version of Livelink WCM Server

Example

The following example shows how to determine the object type with and without using the utility class.

```
ObjectType type;
// without auxiliary class
ObjectHandler oh = VipRuntime.getObjectHandler("DemoWebsite");
type = oh.getWebsite().getObjectType(ObjectType.HTML);
// or
Website w = VipRuntime.getCurrentServer().getWebsite("DemoWebsite");
type = w.getObjectType(ObjectType.HTML);
// with auxiliary class
type = ObjectHandlerUtil.getObjectType("DemoWebsite", ObjectType.HTML);
```

Editing Object Data – The `ObjectData` Interface

By means of the methods of the WCM Java API, you can edit the individual components of WCM objects. The `ObjectData` interface provides the required functions:

- reading and changing standard metadata of WCM objects (see the following section)
- requesting OID and version of WCM objects (see section “Retrieving OID and Version of WCM Objects” on page 124)
- editing references of WCM objects (see section “Editing References” on page 125)
- reading logs (see section “Reading the Log” on page 125)
- editing content (see section “Editing Content” on page 126)
- loading several objects (see section “Loading Several Objects At Once” on page 127)

Reading and Changing Standard Metadata

The methods for reading and changing the standard metadata of WCM objects are defined in the `ObjectData` interface. The object data of the current WCM object or of an older version of the WCM object is retrieved using the `ObjectHandler` interface (method: `get`). The Javadoc on the `ObjectData` interface provides detailed information on the respective methods.

If you want to change attribute values via the `ObjectData` interface, you must consider some points.

The object data of a WCM object is set by means of the corresponding `set` methods. It does not come into effect, however, until the `change` action of the `ObjectHandler` interface is called. In the meantime, another user may also have changed the metadata of the same WCM object and committed the changes by means of the `ObjectHandler.change` method. The current object data has thus become invalid. The `isInvalid` method of the `ObjectData` interface returns `true` in this case. Calling `ObjectHandler.change` using invalid metadata results in a `VipApiException`. The following code sample minimizes the possibility of this error:

```
ObjectData od = ...
od.setTitle("New Title");
od.setTemplate(null);
if (!od.isInvalid())
{
   VipRuntime.getObjectHandler("demosite").change(cid, od, "remark");
}
```

Some attributes can be set to the value `null` (e.g. `setSubtitle(null)`). For the access control list (ACL), this setting has special semantics: `setACL(null)` has the result that the object's own ACL (which possibly exists) is not used any longer. The ACL of the parent WCM object is used instead. If this object does not have an own ACL, the next parent object is

checked. The system checks the complete topic structure until it finds a parent object with an own ACL, if necessary up to the root object as this always has an own ACL. That means that you can use `setACL(null)` to explicitly restore the inheritance mechanism of the ACL.

In addition to the standard metadata, each WCM object has several special attributes, which consist of the following:

- the attributes of the assigned attribute set of the object type used
- the properties of the object category(ies) assigned to the WCM object

To change special attributes, use the `AttributeHandler` interface, see section “Editing Special Attributes” on page 128.

For information on searching for attributes, see the section on “Attribute Filters” on page 134.

Representation as a `RepositoryEntry`

The object data of a WCM object can also be processed as attribute-value pairs. The `RepositoryEntry` class represents a container (`java.util.Map` interface), which maps keys (`Key` interface) to values (`Value` interface). Objects of the `RepositoryEntry` class are of primary importance for the Portal Manager API.

The `RepositoryEntry` class is part of the `de.gauss.vip.repository` package. It is used in the same way as an object of the `Map` interface of the Java 2 SDK. For detailed information on this class, refer to the online documentation.

Use the following methods to represent the object data of a WCM object as a `RepositoryEntry` object and to set the object data by means of the attributes of this object.

```
public interface ObjectData
{
   ...
   public RepositoryEntry getAsRepositoryEntry(Collection attributes,
      String dplSystem);
   public void setFromRepositoryEntry(RepositoryEntry entry);
}
```

The constants for the attributes used are defined in the `FieldNames` interface. If the `getAsRepositoryEntry` method contains `null` as the argument for the attribute set, all attributes in the `RepositoryEntry` object will be set. In particular, all attributes of the object type's attribute set and all properties of the object category are adopted. Setting the object data considers these attributes and properties. Please note that setting all attributes requires some time – if you only need certain attributes, you should supply them explicitly as arguments for the attribute set.

The parameter for the deployment system is only significant for the URL and the path name, as these attributes depend on the deployment system. If these attributes are not required, `null` can be supplied.

Retrieving OID and Version of WCM Objects

A reference to a WCM object is represented by an object of the `ObjectId` class.

Each WCM-managed object has a version represented by an object of the `Version` class (`de.gauss.vip.api.lang`). A version consists of three numbers: a major version, a minor version, and a micro version number. The micro version number changes when a WCM object is changed in the Edit view (check out, change metadata). The minor version number changes when a WCM object is submitted to Quality Assurance; in this case, the micro version number is set to 0. The major version number is incremented at release; in this case, the minor version number is set to 0.

Editing References

Use the `setLinksTo` method of the `ObjectData` interface to add or remove references in the content of a WCM object. A reference can be a WCM-managed object or an external URL to any document.

References are represented by objects of the `Link` interface (`de.gauss.vip.api.lang`). A concrete class that implements this interface and that must be used for the `setLinksTo` method is the `SimpleLink` class (`de.gauss.vip.api.object`).

Reading the Log

Use the `getLastLogEntries` method of the `ObjectData` interface to retrieve the history of a WCM object. The first argument of this method specifies the number of the log entry, while the second argument determines how many entries are to be retrieved. Log entries are always counted backwards.

All entries can be determined as follows:

```
ContextId cid = VipRuntime.getContextHandler().login(userName,
                                                     password);
ObjectData od = VipRuntime.getObjectHandler("demosite").get(cid,
                                              new ObjectId("100"));
List l = od.getLastLogEntries(0, -1);
```

The following example retrieves the last 10 log entries:

```
int last = od.getNoOfLogEntries();
Iterator i = od.getLastLogEntries(last-1, 10).iterator();
while (i.hasNext())
{
   LogEntry e = (LogEntry)i.next();
   System.out.println("Log:"+e.getDate()+" "+e.getUser().getName
      +" "+e.getMessage(Locale.ENGLISH));
}
```

Editing Content

Livelink WCM Server stores the content of WCM objects in a database table. Use the `getContent` method of the `ObjectData` interface to read the stored content. The content is copied to the specified `OutputStream` object. In addition, you can check whether a WCM object has content at all (`hasContent`) and if so, how many bytes the content consists of (`getContentSize`).

The following example reads the content of a WCM object and copies it to the file **test.tmp**.

```
ObjectData od = ...
if (od.hasContent())
{
   FileOutputStream out = new FileOutputStream("test.tmp");
   od.getContent(out);
   out.close();
}
```

The difference between `getContent` and `getCheckOutContent` is that `getCheckOutContent` adapts references used in the content to the deployment system that is specified as an argument. This means that this method returns the content used for editing, e.g. in the Content client.

You can also supply `null` in the deployment system parameter of `getCheckOutContent`. In these cases, the references are not adjusted accordingly. For further information, refer to the Javadoc.

Loading Several Objects At Once

Usually, the `get` method of the `ObjectHandler` interface is used to load `ObjectData` objects. However, if you want to load many objects at once (e.g. for displaying the subordinate objects of a topic in the navigation), you should not load the objects individually by means of the `ObjectHandler.get` method as this affects the performance. This applies in particular to the remote API because each call of `get` causes additional network traffic.

For loading a large number of `ObjectData` objects at once, the interface `ObjectLoader` is provided. This interface offers two methods called `blockLoad` that make it possible to load a large number of `ObjectData` objects by a single `blockLoad` call. The objects to be loaded must be specified in a list by means of their OIDs. Use the method `VipRuntime.getObjectLoader(String websiteName)` to request the `ObjectLoader` interface for a website. Specify the name of the website in the call.

In the following example, the subordinate objects of an `ObjectData` object are loaded.

```
//get ObjectHandler for the website 'InternetSite' ..
ObjectHandler oh = VipRuntime.getObjectHandler("InternetSite");

//load the root object of the website ...
ObjectData odRoot = oh.get(cid, new ObjectId("1") );

//the list contains the OIDs of the objects directly subordinated
// to the root object (root children)
List rootChildren = odRoot.getChildren();

//get ObjectLoader for the website 'InternetSite'
ObjectLoader oL = VipRuntime.getObjectLoader("InternetSite");
```

```
//use blockload to load the ObjectData objects of the child objects
List childrenOD = oL.blockLoad( cid, rootChildren);
```

Editing Special Attributes

The attribute management can be retrieved using `VipRuntime.getAttributeHandler(website)`.

This interface enables you to read all attribute sets and object categories used for a website.

Example

The following code sample shows how to read attribute sets and object categories of a website:

```
//retrieve the AttributeHandler for website 'InternetSite'
AttributeHandler attrH = VipRuntime.getAttributeHandler("InternetSite");

//retrieve a list which contains all available attribute sets for the
website
List attrSets = attrH.getAttributeSets();

//retrieve a list which contains all available object categories for the
website
List categories = attrH.getObjectCategories();
```

Attribute Sets

In Livelink WCM Server, attribute sets are identified by a name and contain so-called attribute definitions. An attribute definition contains all attribute properties, e.g. the unique identifier, the Java class used for the attribute value, or the maximum number of characters for text attributes.

Each WCM object has a set of standard attributes, the standard metadata (see section “Reading and Changing Standard Metadata” on page 122). The standard attributes of a WCM object can be retrieved by means of the `getStandardAttributes` method of the `AttributeHandler` interface. To read the definition of standard attributes, use the `getStandardAttributeDefinition` method.

In addition to the standard attributes, WCM objects can be assigned attributes from attribute sets. The precondition is that the underlying object type is associated with an attribute set. This association is created in the Admin client (see Livelink WCM Server Administrator Manual). Please note that attribute sets can be used only in connection with the object type. Every WCM object of the `HTML` type, for example, can have properties that are only relevant to HTML documents. To determine the attribute set of an object type, use the `getAttributeSet` method of the `ObjectType` interface.

Object Categories

An object category is also a named set of attribute definitions. The term attribute set is not used in this case, however, because object categories are used in another context. An object category can be assigned to every WCM object in a website. In contrast to the attribute set, this assignment has nothing to do with the object type of the WCM object. There may be an “Invoice” object category, for example, that defines typical properties of an invoice as attributes (e.g. “invoice state”). These attributes can have a value for any object type (e.g. for HTML and for Microsoft Word files). Use the `setObjectCategory` method of the `ObjectData` interface to assign an object category to a WCM object.

Formally, an attribute set and an object category do not differ from each other. They have a unique name and contain a list of attribute definitions.

The definition of an attribute is represented by an object of the `AttributeDefinition` interface. An attribute used in Livelink WCM Server is characterized by the following properties:

- the name of the Java class that represents the attribute value (`Value` interface)

  The following classes can be used for attributes: `StringValue`, `DateValue`, `IntegerValue`, `LongValue`, `ListValue`, `BooleanValue`, `SetValue`, `SimpleLink`, `ObjectId`, `DoubleValue`, `Text`, `TimeValue`.
- a unique identifier for the attribute (as an object of the `Key` interface)
- a localized description of the attribute
- the field length limitation (only relevant for `StringValue` classes)

Classes for Attribute Values

All attributes of a WCM object implement the `Key` interface. The attribute values implement the `Value` interface. As shown in the following class diagram, this interface extends the `Comparable`, `Cloneable`, and `Serializable` interfaces of the Java SDK.

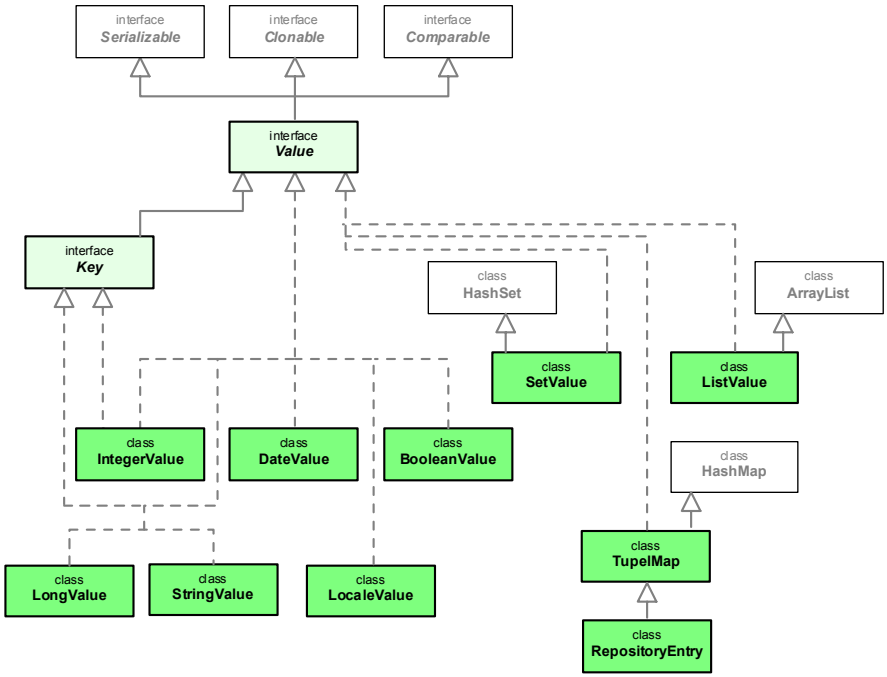


Fig. 10 – Class diagram – `Key` and `Value` interfaces

The object data can also be represented as a container of attribute-value pairs (refer to “Representation as a `RepositoryEntry`” on page 123). Livelink WCM Server also manages attributes from attribute sets and object categories in the form of attribute-value pairs. They are also required for constructing search filters (see “Searching WCM Objects” on page 133).

The `Key` and `Value` interfaces are part of the package `de.gauss.lang`. This package includes concrete implementations of these interfaces, which represent realizations for attribute values. The following list contains the elementary classes used in Livelink WCM Server for the attribute values:

- StringValue
- DateValue
- IntegerValue
- LongValue
- ListValue
- BooleanValue
- SetValue
- Link
- ObjectId
- DoubleValue
- Text
- TimeValue
- LocaleValue

The StringValue class implements the Key interface, among other things.

Searching WCM Objects

The `ObjectHandler.filter` method enables you to search for WCM objects. A search condition is represented as an object of the `Filter` type. These filter objects can be used to model logical expressions that can be evaluated as the Boolean value `true` or `false`. Complex filters can be compiled by means of logical links.

There are basically three different filter class categories.

- filters based on the attributes of the WCM objects (for an overview of the standard metadata, refer to the Javadoc on the `ObjectData` interface, see also section “Editing Object Data – The `ObjectData` Interface” on page 121). These filters extend the abstract `AttributeFilter` class.
- filters that logically link existing filters and thus allow the construction of complex queries. A difference is made between filters with one argument and filters with two arguments (parent class: `UnaryFilter` or `BinaryFilter`). The following filter classes belong to this category: `NotFilter`, `AndFilter`, `OrFilter`.

  > **Note:** It is not possible to use a filter which considers **both** the object position in the topic structure **and** the assigned template, such as `(topic="1") .and. (template="123")`.

- filters that execute a predefined search function: `RootTemplateFilter`, `SubtreeFilter`, and `PrincipalFilter`.

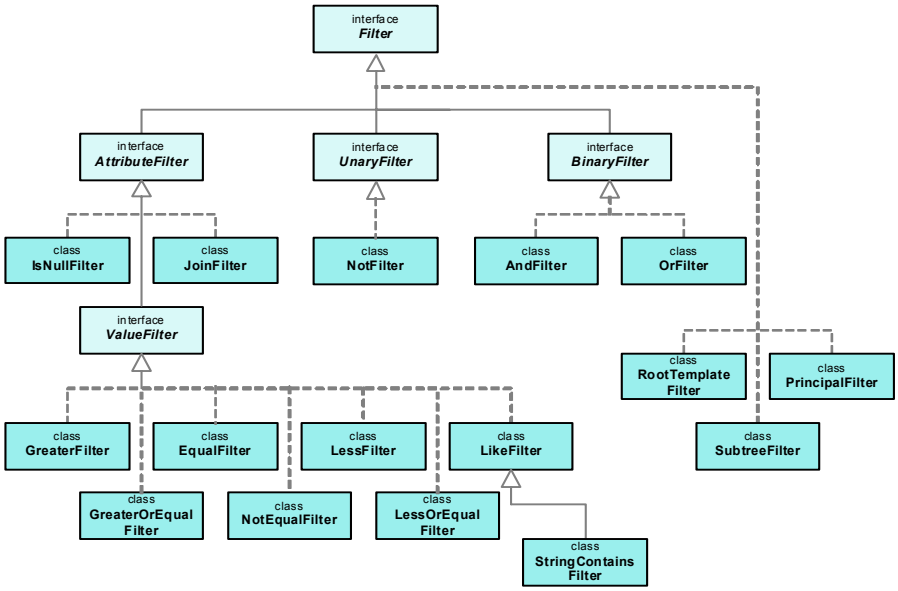


Fig. 11 – Class diagram – Filter classes and interfaces

Attribute Filters

Livelink WCM Server manages the data of a website (including the content and the metadata) in a database. For the search, a query suited for the respective database system (a SQL statement in the form `SELECT ... FROM ... WHERE ... ORDER BY ...`) is formulated on the basis of the filter object. There are, however, limitations regarding the use of attributes for formulating SQL statements. For this reason, not all attributes can be used for searching.

The constants of the attributes are defined in the `FieldNames` and `SearchableKeys` interfaces. Both interfaces are part of the package `de.gauss.vip.api.object`. An overview of the searchable attributes with respective references is contained in the online documentation (Javadoc) on the `ObjectData` interface. In addition to the standard attributes, you can search for user-defined attributes from attribute sets and properties from object categories.

The following table shows which attribute types can be used together with which filters.

Table 11 – Attribute types and filters

	LikeFilter	StringContainsFilter	IsNullFilter[a]	JoinFilter[b]	EqualFilter	NotEqualFilter	GreaterFilter	GreaterOrEqualFilter	LessFilter	LessOrEqualFilter
IntegerValue, LongValue			✓	✓	✓	✓	✓	✓	✓	✓
ListValue[c], SetValue[c]	✓	✓	✓							
DateValue			✓	✓	✓	✓	✓	✓	✓	✓

	LikeFilter	StringContainsFilter	IsNullFilter[a]	JoinFilter[b]	EqualFilter	NotEqualFilter	GreaterFilter	GreaterOrEqualFilter	LessFilter	LessOrEqualFilter
BooleanValue			✓	✓	✓	✓				
StringValue	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
LocaleValue[d], ObjectId, ObjectType, ObjectState, User[d], Version					✓					

(a) This filter can be used to check whether a value has been set for an attribute.

(b) This filter compares two attribute values. Both values must be of the same type.

(c) A list entry is searched which is specified as a `StringValue`.

(d) In the case of `LocaleValue` and `User`, the value can also be specified as a `StringValue`; so the other filters can be used as well.

Example

```
new IsNullFilter(FieldNames.SUBTITLE)

new StringContainsFilter(FieldNames.EMAIL_QA,
      new StringValue("jstein@company.example"))

new EqualFilter(FieldNames.CREATED_BY,
   VipRuntime.getAdminHandler().getUser("jstein"))

new EqualFilter(FieldNames.CREATED_BY, new StringValue("jstein"))
```

Predefined Search Functions as Filters

- The `RootTemplateFilter` class finds all template objects of a website that do not have a template themselves.
- The `SubtreeFilter` class can be used to retrieve all WCM objects below a specified WCM object. This WCM object may be a topic in the topic structure or a template in the template structure. The `SubtreeFilter` can also be used without entering the `Topology` parameter. In this case, the topic structure is considered as default value.
- The `PrincipalFilter` class searches for WCM objects that have been assigned a certain principal (user, group, or role). In addition, it is possible to limit the result to WCM objects for which the principal has a certain right (`VipObjectPermission`). For further information on principals and access rights, see section “Editing Access Control Lists” starting on page 141.

Example

The following filter returns all WCM objects (including the specified start node) that are located below the WCM object with the OID “4711” in the topic structure.

```
new SubtreeFilter(new de.gauss.vip.api.lang.ObjectId("4711"))
```

Additional Search Attributes

The `SearchableKeys` interface extends the `FieldNames` interface by constants that are not directly assigned to an attribute.

These attributes provide additional search functions or make existing search expressions easier to formulate.

ACL_OWNER

Using this attribute, you can determine whether a WCM object has its own access control list (ACL) or whether it has inherited an ACL.

Example

The following filter searches for all WCM objects that have an ACL of their own.

```
new JoinFilter(FieldNames.OID, SearchableKeys.ACL_OWNER)
```

STATE_NAME

This attribute can be used to search for WCM objects with a certain object status.

Example

```
new EqualFilter(SearchableKeys.STATE_NAME,
  new StringValue(ObjectState.RELEASED))
```

is equivalent to

```
new EqualFilter(FieldNames.STATE,
  VipRuntime.getObjectHandler(website)
    .getWebsite().getObjectState(ObjectState.RELEASED))
```

TYPE_NAME

This attribute can be used to search for certain object types.

Example

```
new EqualFilter(SearchableKeys.TYPE_NAME,
  new StringValue(ObjectType.HTML))
```

is equivalent to

```
new EqualFilter(FieldNames.TYPE,
VipRuntime.getObjectHandler(website).getWebsite().getObjectType
    (ObjectType.HTML))
```

Search Parameters

In addition to the filter object, you can specify various other parameters for the search in the `filter` methods. As an example, one of the four `filter` methods is described in the following.

```
public List filter(ContextId cid, Filter filter, ObjectId startOID,
   List sortList, int startResult, int numberOfResults)
   throws VipApiException;
```

- `startOID` – When you specify a start node, the search starts below the specified WCM object. In this case, the start node itself (`startOID`) and all child objects in the topic structure are taken into consideration. The search result is a list of OIDs. The `startOID` must refer to a topic object.

  Tip: To search the entire website, supply `null` as the `startOID`. In this case, the navigation tree is not taken into consideration as all WCM objects are included in the search process anyway. If, on the other hand, the OID of the root node is entered as the `startOID`, the entire navigation tree must be browsed. This process takes considerably longer.

- `sortList` – In this parameter, you can specify a list of attribute constants according to which the result set is to be sorted. By default, the hitlist is sorted by OID.
- `numberOfResults` – This parameter determines the maximum number of WCM objects in the hitlist. The value `-1` returns the entire search result.

Example

The following example searches for all WCM objects that were added after a certain date and have the object status “released” or “delayed release” (`RELEASED` or `PENDING_RELEASE`).

```
public List filterDemo(ContextId cid,String websiteName,Date createDate)
{
   ObjectHandler oh = VipRuntime.getObjectHandler(websiteName);
   Filter f1 = new GreaterFilter(FieldNames.DATE_CREATED,
      new DateValue(createDate));
   Filter f2 = new OrFilter(
      new EqualFilter(SearchableKeys.STATE_NAME,
         new StringValue(ObjectState.RELEASED)),
      new EqualFilter(SearchableKeys.STATE_NAME,
         new StringValue(ObjectState.RELEASE_PENDING));

   Filter f = new AndFilter(f1,f2);
   List l = oh.filter(cid, f, null, null, 0, -1);
   return l;
}
```

Editing Access Control Lists

Each WCM object has an access control list (ACL). The access control list determines which users, groups, and roles may access the WCM object and the rights of these principals. Access control lists are represented by the `Acl` interface (`de.gauss.vip.api.lang`).

An access control list assigns a `Principal` object (user, group, role, group-role, or World) a set of positive (permitted) or negative (not permitted) rights. The rights are represented by objects of the `VipObjectPermission` interface (see table “Constants of the access rights for WCM objects” on page 69).

Use the `getACL` method of the `ObjectData` interface to retrieve the access control list of a WCM object. The `setACL` method changes the access control list. This modification does not take effect until the WCM object is changed using the `ObjectHandler.change` method. To change a WCM object's access control list, a special permission is required (`VipObjectPermission.CHANGE_RIGHTS`).

Access control lists can be inherited: if a certain WCM object does not have its own (object-specific) access settings, the WCM object inherits the access control list of the parent topic. The topic itself can, in turn, have an inherited access control list, etc. The `isInherited` method checks whether the access control list is inherited. In this case, `getOwner` provides the WCM object owning the ACL.

The `checkPermission` method checks whether a given user has a certain right (in regard to the access control list of a WCM object). If access to the user profile is possible, the groups and roles of the user are also checked (the `hasPermission` method only checks the rights directly assigned to the principal). A user logged in via the context management always has a profile (see also “Login and Logout” on page 37). The additional methods are used for adding rights to or removing rights from a given `Principal` object.

Note: It is not possible to supply an object-specific ACL when creating an object. By default, a new object inherits the ACL of the parent topic. You can edit the ACL after the object has been created.

Example

```
//retrieve the AdminHandler and the ObjectHandler for the website
//'demosite'
AdminHandler ah = VipRuntime.getAdminHandler();
ObjectHandler oh = VipRuntime.getObjectHandler("demosite");

//retrieve the ObjectData for the object with the oid 14
ObjectData odObj = oh.get(cid, new ObjectId("14"));

//retrieve the ACL for oid 14
Acl acl = odObj.getACL();

//grant the rights READ and READ_PRODUCTION to the user jstein for oid 14
User jstein = ah.getUserProfile(cid, "jstein",
                                AdminHandler.NO_ASSIGNMENTS);
acl.addPermission(jstein, VipObjectPermission.READ, Acl.GRANT );
acl.addPermission(jstein, VipObjectPermission.READ_PRODUCTION,
                  Acl.GRANT );
odObj.setACL(acl);
oh.change(cid, odObj, "granted read rights for user jstein");

...
```

Performing Workflow Actions

For general information on the workflow in Livelink WCM Server, refer to section “WCM Objects in the Workflow” on page 102. All classes and interfaces described here are included in the package `de.gauss.vip.api.wfm`.

A workflow object contains a workflow definition and is represented by the `Workflow` interface.

The workflow objects of a website are managed by means of the `WorkflowHandler` interface. To request the `WorkflowHandler` via the method `VipRuntime.getWorkflowHandler`, the name of the website must be supplied.

To retrieve the names of all websites of the current Content server, use the method `VipRuntime.getCurrentServer.getWebsiteNames`.

Elements of a Workflow Definition

A workflow definition consists of activities and transitions which implement the `Entity` interface. This ensures a clear assignment of the workflow elements.

Activities

The individual activities of a workflow are represented by the `Activity` interface. This interface extends the `Entity` interface. The activity type represents the staging of Livelink WCM Server.

The following types of activities exist:

Table 12 – Activity types

Activity type	Value	Description	Removes the workflow
`Activity.START`	Start	Start activity of the workflow, corresponds to the assignment of the workflow object to the WCM object	
`Activity.EDIT`	Edit	Corresponds to the object status “changed” or “rejected”	

Activity type	Value	Description	Removes the workflow
`Activity.QA`	QA	Corresponds to the object status “submitted”	
`Activity.DELETE`	Delete	Corresponds to the object status “deleted”	
`Activity.DESTROY`	Destroy	Corresponds to destroying the object	✓
`Activity.RELEASE`	Release	Corresponds to the object status “released”	✓
`Activity.STOP`	Stop	Corresponds to removing the assignment between workflow object and WCM object	✓

Transitions

A targeted transition between two workflow activities is represented by the `Transition` interface. This interface extends the `Entity` interface.

In addition to the name and the description of the transition (via the `Entity` interface), you can retrieve the activities assigned to a transition.

Assigning a Workflow

Both the Content client and the WCM Java API provide functions for assigning workflow objects to other WCM objects.

When a workflow is assigned to a WCM object, a transition from the starting point to an activity must additionally be selected. After the assignment, the target activity of this transition is the current activity. Please consider the following:

- When a workflow is assigned to a WCM object, the latest released version of the workflow object is always used.
- The principal that makes the assignment must have the right “Read” for the workflow object and the WCM object.
- The target activity of the specified transition must conform to the current staging of the object.

Forwarding the Objects in the Workflow

It is not possible to perform normal staging actions on a WCM object to which a workflow is assigned. The object may only be edited by the principal assigned to the current activity. Depending on the principal's access rights, the principal may check out, edit, check in, and move the object, as well as restore an earlier object version.

Having edited the WCM object, the user forwards the object in the workflow. For this purpose, the user selects a predefined transition. This takes the object to the next activity. If this involves a staging transition, the user must have the required access rights.

If an e-mail is to be sent, it is automatically sent to the e-mail address of the principal assigned to the target activity. In the case of staging transitions, no e-mails are sent to the e-mail addresses saved in the metadata of the object.

Unassigning a Workflow

If the WCM object reaches an end activity in the workflow (`Destroy`, `Release`, `Stop`), the assignment between workflow and object is automatically removed.

However, it is possible that an object cannot be forwarded in the workflow, e.g. because the principal assigned to the current activity does not have the required access rights. In this case, you must remove the workflow manually. A principal with the right “Change rights” can change the assignment of a workflow to a WCM object at any time. The principal who assigned the workflow is allowed to unassign it even without the right “Change rights”, provided that the object has not yet been forwarded within the workflow.

Converting Objects

Converters are used to transform contents that exist in a certain file format into other formats. The “Executable” converter is supplied with Livelink WCM Server. This converter calls executable files and supplies the file to be converted as a parameter.

On the basis of the existing interfaces, you can implement custom converters. The following example describes a converter that resizes JPG files to a fixed size and generates an HTML page.

Notes

- The complete code of the sample converter is located in the directory **{WCM installation directory}\examples\converter**.
- The sample converter described here does not catch errors.
- For further information on configuring a converter, refer to the Livelink WCM Server Administrator Manual.

For the WCM system to call an implemented converter, the interface `de.gauss.vip.api.object.ContentConverter` must be implemented. This interface contains only the `convert` method.

The master Content server calls the `convert` method when a WCM object is to be converted that has the file extension specified in the configuration. The call is made synchronously. The object to be converted is locked and released again when the `convert` method returns. For this reason, implementation must be very accurate.

The following method contains the file name of the source file to be converted which the master Content server generated before the method was called.

```
public void convert(
      String source,
      String destination,
      ContentConverterProperty contentConverterProperty)
      throws VipApiException
```

The source file can be directly instantiated as data type “File”.

```
File orgImage = new File(source);
```

In contrast to this, the supplied parameter `destination` contains a file name. When the `convert` has returned, the master Content server tries to find a file with this name. This file is imported into the WCM system. All files that are relatively referenced by this file are also created as WCM objects.

The following code sample illustrates the generation of such a target file:

```
File destHTML = new File(contentConverterProperty.getTempDir(),
      destination);
```

The sample converter generates an HTML file, which contains the reference to the resized image. The file name used is generated from the `source` parameter. The `contentConverterProperty` parameter is a reference to the configuration of the converter. Special parameters (in this example “HEIGHT” and “WIDTH”) can be queried by means `of getEnvironment`. The method returns a list of strings. Each string has the structure “key=value”. By means of this mechanism, it is possible to supply the appropriate configuration to the converter.

The sample converter reads the height and width of the image to be generated. This is realized by the following code sample:

```
private void loadImageSize(List params)
{
      if (params!=null)
      {
         for (int i=0;i<params.size();i++)
         {
            String tempParam=(String)params.get(i);
            if (tempParam.startsWith(WIDTH) &&
                tempParam.length()>WIDTH.length())
            {
               widthM=Integer.parseInt(tempParam.
               substring(WIDTH.length()));
            }
               else if (tempParam.startsWith(HEIGHT) &&
               tempParam.length()>HEIGHT.length())
            {
               heightM=Integer.parseInt(tempParam.
               substring(HEIGHT.length()));
            }
         }
      }
}
```

In a regular converter, the values `getTimeout` and `getMaxSourceLength` should additionally be considered. The timeout specifies a period of time in milliseconds after which the converter must return at the latest (even without a result). The `MaxSourceLength` value should be read by the converter to determine if the source file does not exceed the specified size

in bytes. If this is the case, a simpler or no conversion at all should be performed. Both parameters can be used as additional parameters for the converter. Whether they are considered depends on the converter's implementation.

E-Mail Support

By means of the following objects, e-mails can be created and sent in a WCM system.

The `EMailData` interface represents an e-mail, which is converted into the standard e-mail format when sent to the recipient. The `MailHandler` interface allows access to a new instance of `EMailData` and can send this instance to an SMTP server via a WCM server. In the case of status transitions of objects, an e-mail can be generated by means of the `EMailInfo` class. `EMailInfo` implements `EMailData`, but generates the content of the e-mail itself. Usually, the objects affected by status changes are listed in the sent e-mail.

The following functions are provided:

- specifying several e-mail recipients: `addRecipient`, `addCCRecipient`, `addBCCRecipient`, and `addReplyTo`
- sending e-mails in HTML format: `prepareHTMLMail(String plaintext, String html)`
- adding attachments to the e-mail: `addAttachment(File,...)`, `addAttachment(Object, ...)`, and `addAttachment(InputStream, ...)`
- modifying the text generated by the `EmailInfo` class by means of configuration parameters

- inserting references into the e-mail by means of the `EMailInfoWithLinks` class. If a reference is selected, the respective WCM object is displayed directly in the Content client and can be edited.

To achieve a better compatibility with SMTP servers, the JavaMail API by Sun Microsystems is used for generating the e-mails.

In addition to the already known format for e-mail addresses, e.g.

jstein@company.example

it is possible to specify a name with the e-mail address, e.g.

“Joshua Stein” <jstein@company.example>

Generating E-Mails At Status Changes

When WCM objects change their status, an e-mail can be generated via the `ObjectHandler` interface. For this purpose, an object of type `EMailInfo` is supplied to the respective method, e.g. `ObjectHandler.submit(..., myEMailInfo)`.

The content of the e-mail is generated automatically by `EMailInfo`. By default, the WCM objects that changed their status are listed in the e-mail. Alternately, you can add a predefined text to the e-mail instead of the list. Use the methods `setFixedPlain` or `setFixedHTML` to set this text. The individual methods are described in detail in the Javadoc of the `EMailInfo` class.

It is also possible to use the default text for HTML e-mails and a user-defined text for plain-text e-mails (and vice versa).

Configuring E-Mail

Specifying the content

In the file **{WCM installation directory}\config\mail-conf.xml**, you can configure the content of the e-mails generated automatically by the WCM system. This also includes the e-mails generated by the utility “Release control” and by the server agents “ExpirationNotification” and “ReleaseNotification”.

> **Note:** Custom server agents can optionally request an `EMailInfo` object corresponding to the configuration by calling the method `EMailInfo.CreateConfiguredInstance`.

In the configuration file, you determine whether and in which way e-mails in plain text or HTML are generated. You also define which e-mails are available in Content clients. For the changes to take effect, all servers must be restarted. For further information on the individual configuration parameters, refer to the Javadoc of the `EMailInfo` class.

> **Notes:**
>
> The settings in the configuration file **mail-conf.xml** apply to all websites that are managed by the Administration server.
>
> The encoding of the e-mails sent is limited to ISO-8859-1.

Adapting the representation

There are several possibilities to adapt the representation of the e-mail:

- For both the representation as plain text and in HTML format, either the text predefined in the files **MailMessage_*.properties** (default setting) or a user-defined text can be used (properties `fixedPlain` and `fixedHTML`).

  For detailed information on editing properties files and on localizing GUI texts, refer to the Livelink WCM Server Administrator Manual.

- The predefined representation can be adapted system-wide. For this purpose, the properties file containing the texts must be modified.
- You can derive from the `EmailInfo` class and overwrite the methods `getPlainStart`, `getHTMLStart`, etc. This way, the desired representation can be generated individually for each e-mail (see the second code sample). The `EmailInfoWithLinks` class is an example of this procedure.

Note: For detailed information on the individual keys and values of the properties files, refer to the Javadoc of the package `de.gauss.vip.api.mail`. This Javadoc also contains information on how to adapt the representation of the e-mails by means of the properties files.

Examples

The first code sample shows how an e-mail with a predefined text is sent both in HTML format and in plain text when a WCM object is rejected.

```
EMailInfo anEMailInfo = new EMailInfo("Subject", null, "MessageText",
                                                          true, true);
anObjHandler.reject(aContextId, anObjectIdList, "Remark", anEMailInfo);
```

In the second code sample, a WCM object is submitted to Quality Assurance. An e-mail with attachments is sent. For the representation of the e-mail, a user-defined format is used. This user-defined format is realized by means of the utility class `MyOwnEMailInfo`.

```
EMailInfo em = new MyOwnEMailInfo();
//set a receiver and 2 bcc-receivers
em.addRecipient("user@company.example");
String mailBCC = "testuser@company.example,
      \"User 2\" <u2@company.example>";
em.setBCCRecipients
      (VipRuntime.getMailHandler().separateEMailAddresses(mailBCC));
//set content: generate htmlText using the messageText
//and use fixed plainText
```

```
em.setSubject("a subject");
em.setMessageText("a message");
em.setSendHTML(true);
em.setFixedPlain("Please upgrade your mailclient.");
//add some attachments
em.addAttachment(new File("test_file.xml"), "filename_in_mail.txt");
em.addAttachment("some text to read", "document.txt");
//and finally submit the object and send the e-mail.
aObjectHandler.submit(aContextId, aObjectIdList, aDate, "a remark", em);
```

Sample code of the utility class `MyOwnEMailInfo`:

```
public class MyOwnEMailInfo extends EMailInfo {
      public String getHTMLStart(String remark, Locale loc) {
         return "<HTML><HEAD></HEAD><BODY>" + remark
            + "<TABLE><TR><TD>OID</TD><TD>Title</TD></TR>\n";
      }
      public String getHTMLForObject(ObjectData od, Locale loc) {
         try {
            String oid = od.getObjectId().toString();
            String title = od.getTitle();
            return "<TR><TD>" + oid + "</TD>" + "<TD>" + title
            + "</TD></TR>\n";
         } catch (VipApiException e) {
            return "Exception:" + e;
         }
      }
      public String getHTMLEnd(String remark, Locale loc) {
         return "</TABLE></BODY></HTML>\n";
      }
}
```

Generating E-Mails via the WCM Java API

E-mails cannot only be generated in the case of status changes of WCM objects, but also by JSP pages, server agents, and standalone Java programs accessing the remote API.

First, the e-mails are transmitted to a WCM server (in the case of a JSP page or a server agent, this may be the same server). This server forwards the e-mails to an SMTP server.

Use the `VipRuntime.getMailHandler` method to access the `MailHandler` interface. By means of the `createEMailData` method, you can generate an object of type `EmailData`.

Compiling the E-Mail

In the next step, the e-mail is compiled. There are different methods for setting the fields “Subject”, “Sender”, and “Reply-To”. The methods `addRecipient`, `addCCRecipient`, and `addBCCRecipient`add recipients to the fields “To”, “CC”, and “BCC”.

To set the content of the e-mail and - optionally - a MIME type, use `setBody`. To send an e-mail in HTML format with an alternative representation as plain text, the method `prepareHTMLMail(String plainText, String htmlText)` must be called. In this case, e-mail clients that do not support HTML get the plain text. In e-mail clients that do support HTML, the text is displayed in HTML format.

To attach files to the e-mail, use the method `addAttachment(File fileObj, String nameInEMail, String mimeTyp)`. You may omit the `mimeTyp` parameter; it is automatically determined on the basis of the file extension before the e-mail is sent to the SMTP server. By means of the method `addAttachment(Object o, ...)`, the output of the method `o.toString` can be used as the data basis for the attachment. By means of the method `addAttachment(InputStream, ...)`, any input stream can be added as an attachment.

Sending the E-Mail

The `EmailData` object compiled in the previous step can now be sent from the current WCM server to an SMTP server by means of the method `MailHandler.sendEMail`. A valid context ID must be specified. The context does not require any further rights, it must only be valid on the sending WCM server. Optionally, a different available WCM server or the SMTP host and port can be specified.

Example

The following code sample shows how to send an e-mail with attachment:

```
EMailData md = VipRuntime.getMailHandler().createEMailData();
md.setSender("sender@company.example");
md.setSubject("a subject");
md.setBody("a bodytext");
md.addRecipient("recipient@company.example”);
md.addAttachment(new File("test_file.xml"), "filename_in_mail.txt");
VipRuntime.getMailHandler().sendEMail(aContextId, md);
```

CHAPTER 6

System Information and Administration

By means of the functions of the WCM Java API, you can retrieve information on different components of the WCM system and on the status of the deployment. Also, you can change the system status, and use the report and tracing functions of the servers. For this purpose, the following interfaces are provided: `AdminHandler`, `SystemHandler`, and `DeploymentHandler`.

Retrieving information on the WCM system

- retrieve the users that are logged in to the WCM system (see “Retrieving Logged-in Users” on page 160)

  `SystemHandler` interface

- retrieve information on the servers used (see “Information on Servers” on page 160)

  `AdminHandler` interface

- retrieve information on websites (see “Information on Websites” on page 162)

  `AdminHandler` interface

Retrieving information on the deployment systems

- retrieve a list of all deployment systems (see “Retrieving a List of Deployment Systems” on page 165)

  `AdminHandler` interface

- retrieve deployment metadata (e.g. URL and path) for a generated page within a deployment system (see “Deployment Metadata” on page 166)

  `DeploymentHandler` interface

- retrieve status information on a deployment system and on running transactions (see “Information on the Deployment Status” on page 167)

  `DeploymentHandler` interface

- retrieve error lists and error messages that occurred during a running transaction within a deployment system as well as error messages related to a certain WCM object within a deployment system (see “Deployment Errors” on page 171)

  `DeploymentHandler` interface

Changing the system status (run levels)

Retrieve and set run levels (see section “Retrieving and Setting Run Levels” on page 173)

`AdminHandler` interface

Using reports, logs, and tracing for locating errors

- read the logs of the servers (see “Server Logs” on page 174)

  `SystemHandler` interface

- access the reports of the servers (see “Server Reports” on page 175)

  `SystemHandler` interface

- use the tracing functions (see “Using Tracing Functions” on page 176)

  `SystemHandler` interface

> **Note:** For accessing the system administration functions, the administration right “Access to system administration” is required. For further information, refer to section “Administration Rights of Principals” on page 65.

You can get an instance of the `SystemHandler` interface via `VipRuntime.getSystemHandler`, an instance of the administration interface via `VipRuntime.getAdminHandler`, and an instance of the `DeploymentHandler` interface via `VipRuntime.getDeploymentHandler`.

Unless otherwise specified, all classes and interfaces are included in the package `de.gauss.vip.api.admin` and `de.gauss.vip.api.deployment`.

Retrieving Information on the WCM System

The interfaces `AdminHandler` and `SystemHandler` provide methods for retrieving information on the WCM system. This includes, among other things, the logged-in users, the installed servers, and the configured websites.

Retrieving Logged-in Users

The system administration (`SystemHandler` interface) offers an exact overview of the users that are currently logged in to the WCM system. Use the `getActiveUsers` method for this purpose. If a server name is specified in the request, only the users logged in to this server are returned. If `null` is specified as the server name, all users logged in to the WCM system are listed.

> **Note:** Via the `AdminHandler` interface, you can create and edit principals and retrieve their profiles (see chapter 3 “User Administration”).

Information on Servers

A WCM system consists of a set of WCM servers that communicate with each other. For detailed information on the server concept of Livelink WCM Server, refer to chapter “Concepts” of the Livelink WCM Server Administrator Manual

Each server is clearly identified by a logical name and has a type that defines the server’s role (Administration server or Content server). Server type and name can be retrieved by means of the `Server` interface.

This interface also provides the `isMaster` method, which can be used to check whether the server is a master server (master Content or master Administration server). Only a master server has write access to the data of the WCM system. If the method returns `false`, the server is a proxy server. A proxy server merely has read access to the WCM objects of a website or the administration and configuration information. It delegates all actions that require write access to a master server.

Detailed server information can be obtained via the `AdminHandler` interface. Use the `getServer` method for this purpose. It returns a `Server` object which contains the following server properties:

- the Administration server responsible for user administration and system configuration
- storage method of the user data (RDBMS, Livelink, or LDAP)
- the server's host name and the port numbers used for communication
- the host name of the mail server which is used for sending automatic notifications, and the sender's e-mail address
- the languages offered by the server for localizing messages
- the list of all websites managed by this server or to which this server has read access (in the case of proxy Content servers)
- the list of the deployment systems installed on this server
- the installation directory as well as the server's temporary directory. In the server's temporary directory, there is a temporary directory for each website. This directory is required for object management (e.g. for checking out WCM objects).

Example

The following code sample shows how to read server information:

```
//log in user...
ContextId cid = VipRuntime.getContextHandler().login("..." , "...");

//retrieve AdminHandler...
AdminHandler ah = runtime.getAdminHandler();

//retrieve Admin server
Server admin = ah.getServer(cid, "Admin");

//print out some information about the server
System.out.println( admin.getHostname() );
System.out.println( admin.getSocketPort() );
System.out.println( admin.getHTTPPort() );
System.out.println( admin.getMailHost() );
...
```

Information on Websites

In a WCM system, a website is clearly identified by a logical name. The `AdminHandler` interface provides methods for retrieving information about a website. A website is modeled by an object of the `Website` class. The `getWebsite` method of the `AdminHandler` interface provides the essential website properties. The list of all websites in a WCM system is determined by means of `getWebsites`. The result is a list of `Website` objects.

The following properties can be determined for a `Website` object:

- the website's root object, i.e. the WCM object that represents the root of the navigation tree
- the set of all users, roles, and groups that are assigned to this website. Only users with a corresponding assignment (a direct assignment as a user or an indirect assignment based on the user's group or role membership) have access to this website. The sets contain respective objects of the classes `User`, `Role`, and `Group`.

- the name of the website's master Content server as well as the name of the JDBC pool used by the master Content server for saving the website data
- a list of proxy Content servers (`String` objects) that have read access to this website (and delegate write accesses)
- a list of all object types that are defined for this website. The list contains objects from the `ObjectType` interface.
- the list of all object statuses with objects of the `ObjectState` interface
- the encoding used for generating the pages
- a list of deployment systems (`String` objects) for this website
- the routing information of the website
- the temporary website directory. This directory is used for files to be written by the server, e.g. to save the content of a checked-out WCM object.

Example

The following code sample shows how website information is read:

```
//log in user...
ContextId cid = VipRuntime.getContextHandler().login("..." , "...");

//retrieve AdminHandler ..
AdminHandler ah = runtime.getAdminHandler();

//retrieve website 'InternetSite' with all indirect assignments ( flag
//INDIRECT_ASSIGNMENTS..)
Website ws = ah.getWebsite(cid, "InternetSite",
AdminHandler.INDIRECT_ASSIGNMENTS);

//print out some information about the website ....
System.out.println( ws.getNavigationRoot() );
System.out.println( ws.getProxyServers() );
```

```
System.out.println( ws.getTemporaryDirectory() );
System.out.println( ws.getDeploymentSystems("Master-Content") );
...
```

Information on the Deployment

From the WCM objects stored in the database, the deployment systems generate files that can be displayed by means of a browser. These objects are called pages. Using the functions of the WCM Java API, you can retrieve a list of deployment systems, as well as data on the status of the deployment.

Use the `AdminHandler` interface to access the list of deployment systems (see the following section).

To access deployment metadata and status information, use the `DeploymentHandler` interface. It offers the following functions:

- retrieving deployment metadata (e.g. URL and path) for a generated page within a deployment system

  See section “Deployment Metadata” on page 166

- retrieving status information on a deployment system and on running transactions within a deployment system

  See section “Information on the Deployment Status” on page 167

- retrieving error lists and error messages that occurred during a running transaction within a deployment system, as well as error messages related to a certain WCM object within a deployment system

  See section “Deployment Errors” on page 171

You can get an instance of the `DeploymentHandler` interface via `VipRuntime.getDeploymentHandler`. Unless otherwise specified, all classes and interfaces are included in the package `de.gauss.vip.api.deployment`.

Retrieving a List of Deployment Systems

Using the `getDeploymentSystems` method of the `AdminHandler` interface, you can retrieve the list of all deployment systems. The method returns a list of deployment systems as objects of type `DeploymentSystem`.

The essential properties of a deployment system are:

- name
- type (`EDIT`, `QA`, `PROD`)
- category (`STANDARD`, `STANDARD_FLAT`, `DYNAMIC`, `WEBDAV`, `CONTENT`, and `INSITE_EDIT`)
- the website the deployment system is based on
- the server on which the deployment system is executed
- the directory on the server for storing the generated pages
- the base URL used for generating references

Example

The following code sample shows how to read deployment system information:

```
//log in user...
ContextId cid = VipRuntime.getContextHandler().login("..." , "...");

//retrieve AdminHandler..
AdminHandler ah = runtime.getAdminHandler();
```

```
//retrieve a list of all available deployment systems, includes the
//deployment systems of all websites..
List dplSys = ah.getDeploymentSystems(cid);

//retrieve a deployment system by name
DeploymentSystem ds = ah.getDeploymentSystem( cid, "InternetSite_edit");
...
```

Deployment Metadata

The deployment metadata include, for example, the URL and the path to the generated page. Depending on the number of deployment systems installed, several pages are created for each WCM object. The deployment metadata of these pages differ from each other because the pages can be generated for different data storage views (Edit, QA, or Production). Moreover, the servers, to which the deployment systems belong, may be located on different computers.

You can use the method `get` of the `DeploymentHandler` interface to request the deployment metadata of a page. In addition to the context ID of a logged-in user, the OID of the associated WCM object and the name of the deployment system (as a string) are supplied. The deployment metadata returned are represented by the `DeploymentMetaData` interface.

By means of the methods of the `DeploymentMetaDate` interface, you can retrieve the URL and the path to the generated page and, if existing, to the surrogate page or the statified page. Whether a surrogate page is generated depends on the object type. To configure statification for a deployment system, use the Admin client.

If the object does not have a surrogate, both `getSurrogateURL` and `getSurrogatePath` return `null`. The methods `getStaticURL` and `getStaticPath` return `null` if the object has not been statified.

The following example shows how to retrieve the metadata of the root object of a website.

```
//log in
ContextId cid = VipRuntime.getContextHandler().login( ... );

//get DeploymentHandler
DeploymentHandler deplH
               = VipRuntime.getDeploymentHandler("InternetSite");

//get and output deployment metadata for the root object (OID=1)
ObjectId oidRoot = new ObjectId("1");
DeploymentMetaData rootDplMD = deplH.get(cid, "InternetSite_edit",
                                         oidRoot);
System.out.printnln(" root obj URL  : " + rootDplMD.getURL() );
System.out.printnln(" root obj Path : " + rootDplMD.getPath() );
```

Information on the Deployment Status

The `DeploymentHandler` can be used to retrieve status information on deployment systems and the transactions running in the deployment systems. Status information is encapsulated in the `DeploymentStatus` interface.

Note: The information that can be retrieved via the `DeploymentStatus` interface is available in the Admin client via *System administration* → *Active servers* → *{server name}* → *Reports* → *DeploymentSystemHandler*. For information on the contents of the report, refer to the Livelink WCM Server Administrator Manual.

The DeploymentStatus interface can be retrieved via the getStatus method of the DeploymentHandler interface. In addition to the context ID of a logged-in user and the name of the deployment system, the transaction ID of a transaction for which the status information is to be retrieved must be specified when the method is called. If null is supplied instead, the DeploymentStatus object returned contains information on all running transactions of the deployment system.

> **Note:** If a Progress object has been requested for a transaction, the transaction ID can also be retrieved via this object (see “Watching the Progress of Transactions” on page 115). Alternately, the IDs of all transactions running in a deployment system can be retrieved via the method getRunningTransactions of the DeploymentStatus interface.

Via the DeploymentStatus interface, you can retrieve the following information on a deployment system:

- status information on the order analysis
- status information on the execution of deployment job

Order analysis

After each change to WCM objects, the deployment systems must generate new pages that reflect the current state of the WCM objects. For this purpose, a deployment order is created whenever a WCM object is changed. Analyzing this order comprises the following tasks:

- The system checks whether the deployment metadata of the respective page (URL, path, etc.) must be modified.
- The system checks whether other pages are affected by the change (e.g. if a template has been edited).

- For all pages to be generated, so-called deployment jobs are created, i.e. commands for generating a page within a deployment system (dynamic deployment systems are the exception, because they do not generate the pages until they are requested by the HTTP server).

Executing the deployment jobs

In this phase, the created jobs are processed and the pages affected by the changes to the WCM object are regenerated. The generated files (HTML pages, images, etc.) are saved in the file system of the server on which the deployment system is installed.

Available status information

For both phases of the deployment, the following information can be retrieved via the `DeploymentStatus` interface.

- number of orders and jobs waiting to be processed: `public long getNoOfWaitingOrders` and `public long getNumberOfWaitingJobs`
- number of analyzed orders and processed jobs: `public long getNoOfAnalysedOrders` and `public long getNoOfExecutedJobs`
- average processing time for an order or a job: `public double getAvgOrderServingTime` and `public double getAvgJobServingTime`
- average waiting time for an order or a job: `public double getAvgOrderWaitingTime` and `public double getAvgJobWaitingTime`

If you request the deployment status for a certain transaction, i.e. specify a transaction ID when calling `DeploymentHandler.getStatus`, you can also retrieve lists with the OIDs of the WCM objects whose pages are being analyzed/processed in this transaction. Use the methods `getAnalysingOidList` and `getExecutingOidList` for this purpose. Moreover, messages (as strings) describing the progress of the analysis or job execution can be called for the transaction. For this purpose, the methods `getAnalysingMessage` and `getExecutingMessage` are used.

If the deployment status was requested without a transaction ID, the methods return `null`. After a transaction has been completed, status information is no longer available.

Example

The following example shows how to return the OIDs of the WCM objects whose pages are currently being analyzed/processed. The associated messages are also requested. The status information refers to all actions running in a deployment system, i.e. no transaction ID is supplied.

```
//log in and get DeploymentHandler
ContextId cid = VipRuntime.getContextHandler().login( ... );
DeploymentHandler dH = VipRuntime.getDeploymentHandler("InternetSite");

//get DeploymentStatus interface for the complete deployment system,
//i.e. not for a specific transaction
DeploymentStatus ds = dH.getStatus(cid, "InternetSite_edit", null);

//get list of running transactions. Iterate through this list
//to get the DeploymentStatus objects for the individual transactions
List runningTransactions = ds.getRunningTransactions();
Iterator transactionIter = runningTransactions.iterator();
while ( transactionIter.hasNext() )
{
   Long transactionID = (Long) transactionIter.next();

   //get DeploymentStatus for the individual transactions and
   //output information
   DeploymentStatus dsTransaction = dH.getStatus(cid,
                                    "InternetSite_edit", transactionID);
   System.out.println( "oids being analysed in Transaction "+
                       dsTransaction+ " : " +
```

```
                    dsTransaction.getAnalysingOidList() );
   System.out.println( dsTransaction.getAnalysingMessage(Locale.
                                                    getDefault() ) );
   System.out.println( "oids being executed in Transaction "+
                    dsTransaction+ " : " +
                    dsTransaction.getExecutingOidList() );
   System.out.println( dsTransaction.getExecutingMessage(Locale.
                                                    getDefault() ) );
}
```

Deployment Errors

The `DeploymentHandler` interface can also be used to retrieve information on errors that occurred during the deployment. For this purpose, the following methods are available:

- `public List getErrors(ContextId cid, String dplName, ObjectId oid)` – This method provides information on errors that occurred in the specified deployment system while processing a certain object. The object is specified by means of its OID.

  The method returns a list containing the errors as objects of the type `DeploymentError`.

- `public List getErrors(ContextId cid, String dplName, Long transactionId)` – This method provides information on errors that occurred in the specified deployment system during a certain transaction. The transaction is specified by means of a transaction ID.

  The method returns a list containing the errors as objects of the type `DeploymentError`.

- `public List getErroneousTransactionIds(ContextId cid, String dplName)` – This method provides information on errors that occurred in the specified deployment system.

  The method returns a list with the IDs of the transactions during which errors occurred.

By means of the methods of the `DeploymentError` interface, you can retrieve the OID of an object or the ID of a transaction in which the respective error occurred (methods `getOID` and `getTransactionId`). Moreover, the actual error message can be requested by means of the method `getErrorMessage`. The method `getCreationDate` retrieves date and time of the error.

Example

The following example shows how to retrieve the deployment errors for the currently running transaction.

```
//log in and get DeploymentHandler
ContextId cid = VipRuntime.getContextHandler().login( ... );
DeploymentHandler dH = VipRuntime.getDeploymentHandler("InternetSite");

//get DeploymentStatus for the deployment system in order
//to retrieve the running transaction
DeploymentStatus ds = dH.getStatus(cid, "InternetSite_edit", null);
List runningTransactions = ds.getRunningTransactions();

//iterate through the running transactions in order to
//output deployment errors
Iterator transactionIter = runningTransactions.iterator();
while ( transactionIter.hasNext() )
{
   Long transactionID = (Long) transactionIter.next();

   //get list with deployment errors of the current transaction
   //iterate through the list of DeploymentError objects
   //output messages
   List dplErrors = dH.getErrors(cid, "InternetSite_edit",
                                 transactionID);
   Iterator dplErrIter= dplErrors.iterator();
   while (dplErrIter.hasNext())
   {
```

```
        DeploymentError dplErr=(DeploymentError) dplErrIter.next();
        //output OID and error message
        System.out.println(" deployment error occured for oid " +
                    dplErr.getOID()+ " in transaction " + transactionID );
        System.out.println(" error message : "+
                      dplErr.getErrorMessage(Locale.getDefault()) );
    }
}
```

Retrieving and Setting Run Levels

The run level describes the current state of a server or a website. It defines which functional components of the server or the website are currently active or inactive. The run levels are divided into server run levels and website run levels. The website run levels are based on the server run levels.

The possible states of a server or a website are defined in the `Runlevel` interface. When a server starts, the run level progresses from `SERVER_DOWN` to `WEBSITE_UP`, when the server shuts down, from `WEBSITE_UP` to `SERVER_DOWN` accordingly. The run levels of servers and websites can also be controlled separately. For further information on run levels, refer to the Livelink WCM Server Administrator Manual and to the Javadoc on the `Runlevel` interface.

The administration interface `AdminHandler` provides methods for changing the run level of the current server or website, e.g. in order to shut down the system to back up the website. The `getRunlevel` method is used to retrieve the run level of the server or website. The `setRunlevel` method changes it accordingly. This is usually a time-consuming process (similar to starting up or shutting down an operating system).

> **Note:** For information on the events fired in the case of run level changes, refer to section “System-Related Events” on page 85.

Logs, Reports, and Tracing

Each server creates several logs. These logs contain information and error messages of the servers. The reports of the servers contain messages on the actions in the WCM system. Both reports and logs can be used to analyze and solve problems.

Tracing is another means of locating errors. By means of the tracing function of Livelink WCM Server, you can create a log with the running actions and messages of a server. Additionally, it is possible to trace individual classes via class-specific filters.

The `SystemHandler` interface provides access to server logs, server reports, and tracing functions.

Server Logs

The logs of the servers are stored in the directories **\log**, **\log\deployment**, and **\log\contentminer** of the WCM installation directory on the computer hosting the respective server. The individual logs can be viewed via the Admin client (see Livelink WCM Server Administrator Manual).

The logs can also be accessed via the WCM Java API. The `SystemHandler` interface provides methods for this purpose. By means of the method `public List getLogFiles`, for example, you get a list of the log files belonging to a specified server. The content of an individual log file can be retrieved by means of the method `public String getLogFile` by specifying the file name and the server.

The following example first retrieves a list with the names of the log files and then calls the content of the individual log files.

```
// get SystemHandler
SystemHandler sysHandler = VipRuntime.getSystemHandler();
```

```
// load list with log file names of the current server
Server currentServer = VipRuntime.getCurrentServer();
List logFiles = sysHandler.getLogFiles(cid, currentServer );

//iterate through the result
Iterator logFileIter = logFiles.iterator();
while ( logFileIter.hasNext() )
{
      String logFileName = (String) logFileIter.next();
      //get logs
      int logFileNumOfLines = sysHandler.getLogFileNumOfLines(cid,
                                          currentServer, logFileName );
      String[] logFileContents = new String[logFileNumOfLines];
      logFileContents = sysHandler.getLogFile(cid, currentServer,
                                            logFileName, 0);
...
}
```

Note: The trace logs of a server can be read in the same way.

Server Reports

The system automatically creates reports for the various software components of the servers. These reports reflect the current status of the WCM system. Depending on the type and configuration of the server, different reports are created. For example, deployment reports are only available for servers on which deployment systems have been set up. You can view the reports in the Admin client (see Livelink WCM Server Administrator Manual).

The `SystemHandler` interface provides the method `public List getCoreReport` for requesting a specific report from a server. The name of the report is supplied as a string. The returned list contains the individual lines of the report as string objects. If `null` is supplied instead of the report name, this method returns the names of all report objects available on this server as a list of strings.

Note: For retrieving the report of the CustomPoolManager (on custom connections in the WCM system), the respective custom connection must provide the required information (see “Reports for Custom Connections” on page 47).

The following example first retrieves a list with the report names and then reads the content of the reports.

```
// get SystemHandler
Server currentServer = VipRuntime.getCurrentServer();
SystemHandler sysHandler = VipRuntime.getSystemHandler();

// load list with report names of the current server
List repObjects = sysHandler.getCoreReport(cid, currentServer, null);

//iterate through the result in order to get the reports
Iterator repIter = repObjects.iterator();
while ( repIter.hasNext() )
{
      String report = (String) repIter.next();
      List reportContents = sysHandler.getCoreReport(cid, currentServer,
                                                     report);
.....
}
```

Using Tracing Functions

The `TraceFilter` interface provides access to the tracing functions.

For tracing a server or a class, you can determine exactly which events are to be logged in the tracing. The `TraceFilter` interface provides constants for this purpose. The following table explains the meaning of the individual constants:

Table 13 – Constants for tracing settings (`TraceFilter` interface)

Constant	Function
`METHOD_ENTER`	Logs the entry into the methods of the classes used in the WCM system
`METHOD_LEAVE`	Logs the exit from a method
`FATAL_ERROR`	Logs errors with the level “Serious”, which, for example, result in the respective server being no longer operable
`ERROR`	Logs errors with the level “Error”, which occur during actions in the WCM system. Errors of this level usually do not result in the server no longer being operable.
`WARNING`	Logs errors with the level “Warning”
`INFO_MESSAGE`	Logs information messages

Tracing can be performed globally for the complete server or restricted to one or several classes (class-specific tracing).

By default, tracing is deactivated. To activate the global tracing, use the `setGlobalTraceFilter` method. For class-specific tracing, the method `setClassTraceFilter` is used. The respective tracing settings are supplied in the form of a list with `TraceFilter` constants.

In the case of class-specific tracing, tracing must additionally be activated by means of the method `setAcceptClassTraceFilter` after `setClassTraceFilter` has been called. This method can also be used to switch off tracing again. The tracing settings are stored.

The following example activates a global tracing that logs the entry into and exit from methods as well as serious errors.

```
// get SystemHandler
Server currentServer = VipRuntime.getCurrentServer();
SystemHandler sysHandler = VipRuntime.getSystemHandler();

// create list with trace settings
List traceFilter = new ArrayList();
traceFilter.add(TraceFilter.METHOD_ENTER);
traceFilter.add(TraceFilter.METHOD_LEAVE);
traceFilter.add(TraceFilter.FATAL_ERROR);

// activate global tracing for the current server
sysHandler.setGlobalTraceFilter(cid, currentServer, traceFilter);
```

For reading the content of the trace log, use the method `getTraceFile`. To completely delete the trace settings for a server and thus deactivate tracing, supply `null` instead of the list in the method `setGlobalTraceFilter`.

CHAPTER 7

Accessing Livelink Systems

The WCM Java API enables access to Livelink systems. By means of the appropriate methods, you can

- retrieve Livelink principals, see section “Livelink User Administration” on page 183
- read, add, modify, and delete Livelink objects, see section “Livelink Object Management” on page 184
- add Livelink objects as WCM objects to a WCM-managed website and modify these objects, see section “Relators” on page 196

The package `de.gauss.livelink` and its subpackages contain the classes for Livelink access.

Notes

- The classes and interfaces for the Livelink integration are implemented on the basis of the Livelink API (LAPI for short). The constants used are based on the LAPI constants. For more information, refer to the LAPI documentation.
- The Livelink integration provides most of the LAPI functions. For more information, refer to the Javadoc of the WCM Java API.
- You cannot access Livelink systems via the remote API.

The `LivelinkRuntime` class

The `LivelinkRuntime` class is the central class for the Livelink integration. This class provides the point of access to the interface `LivelinkAdminHandler` (access to the Livelink user administration) and to the interface `LivelinkObjectHandler` (access to the Livelink object management).

```
LivelinkAdminHandler llah =
      LivelinkRuntime.getInstance().getAdminHandler(systemName);
LivelinkObjectHandler lloh =
      LivelinkRuntime.getInstance().getObjectHandler(systemName);
```

Enter the parameters of the Livelink systems, which Livelink WCM Server is to access, in the **livelink-systems.xml** file. This file is located in the directory **{WCM installation directory}\config**. Refer to the Livelink Integration Manual for detailed information about this file. By means of `LivelinkRuntime`, you can retrieve all connected Livelink systems and their configuration.

The following code sample retrieves the connected Livelink systems and outputs the name of the Livelink system, as well as the name of the Livelink pool used for the connection.

```
LivelinkRuntime ll = LivelinkRuntime.getInstance();
for (Iterator i = ll.getSystemNames().iterator(); i.hasNext(); )
{
      String systemName = (String)i.next();
      System.out.println("System "+systemName+" uses Livelink pool "
         +ll.getSystemConfiguration(systemName).getPoolName());
}
```

Livelink User Administration

The `LivelinkAdminHandler` interface provides methods for reading and searching Livelink principals.

The users and groups managed in Livelink are represented by according interfaces.

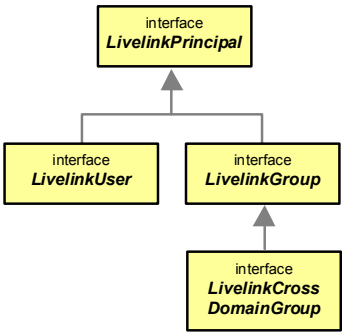


Fig. 12 – Class diagram – Livelink principals

`LivelinkAdminHandler` versus `AdminHandler`

The `LivelinkAdminHandler` wraps the functions of the Livelink API (LAPI for short) for administering users and groups. In parallel, the `AdminHandler` interface provides access to the users, groups, and roles of the WCM system.

Use the `AdminHandler` functions for managing principals if the WCM users are stored in Livelink, i.e. if both systems use a common user administration. If this is not the case, use the `LivelinkAdminHandler` for accessing Livelink users.

Examples for using the `LivelinkAdminHandler` interface

The following code sample retrieves the user with the ID “jstein” and outputs the user's last name.

```
ContextId cid = VipRuntime.getContextHandler().login("admin","admin");
LivelinkAdminHandler llah =
      LivelinkRuntime.getInstance().getAdminHandler(systemName);
LivelinkUser lluser = (LivelinkUser)llah.getUser(cid, "jstein");
System.out.println("lastName="+lluser.getLastName());
```

`LivelinkPrincipalFilter` is the basic class for searching principals in Livelink. Use the `LivelinkUserFilter` class for searching users and the `LivelinkGroupFilter` class for searching groups.

The following code sample searches all groups whose names start with the letter „n“.

```
ContextId cid = ...
LivelinkAdminHandler llah = ...
LivelinkGroupFilter f = new LivelinkGroupFilter("n",
      LivelinkPrincipalFilter.OP_STARTSWITH);
for (Iterator i = llah.filter(cid, f, 100).iterator(); i.hasNext(); )
{
      LivelinkGroup g = (LivelinkGroup)i.next();
      System.out.println("Found group : "+g.getName()+" ("+g+")");
}
```

Livelink Object Management

The `LivelinkObjectHandler` interface provides methods for reading and editing Livelink objects. This section deals with the following topics:

- accessing Livelink objects, see the following section

- checking the permissions for Livelink objects, see section “Checking Permissions for a Livelink Object” on page 189
- supported Livelink object types, see section “Livelink Object Types” on page 189
- retrieving the metadata of Livelink objects, see section “Retrieving a Livelink Object’s Metadata” on page 191
- reading the logs of Livelink objects, see section “Reading Log Entries” on page 192
- retrieving the content of Livelink objects, see section “Reading Content” on page 192
- actions on Livelink objects: adding, deleting, editing, and reserving Livelink objects, see sections “Performing Actions on a Livelink Object” on page 194

Accessing Livelink Objects

Identifying Livelink objects

Accessing Livelink objects presupposes that the objects can be clearly identified. In Livelink, each object is identified by means of its node ID. For accessing the Livelink object, the WCM Java API uses the so-called Livelink ID which is generated from the node ID. For this purpose, the `LivelinkId` class is used. To refer to a certain version of the Livelink object, this version must be specified when generating the Livelink ID. If no version is specified, the ID refers to the current version of the Livelink object.

Additionally, some constructors require the volume ID. If you do not know this ID, you can specify `0` (for more information, refer to the LAPI documentation).

The following example illustrates how to use the different constructors.

```
// reference to node 123
LivelinkId llid1 = new LivelinkId(123);
// reference to node 123 and version 2 with unspecified volume ID
LivelinkId llid2 = new LivelinkId(0,123,2);
// same as above but with volume ID 2000
LivelinkId llid3 = new LivelinkId(2000,123,2);
```

Reading Livelink objects

Use the `LivelinkObjectHandler` interface for reading Livelink objects. The Livelink ID is used to identify the objects.

The following code sample reads a current and a versioned Livelink object. If the logged-in user does not have the permission “See Contents” for the object, a `LivelinkAccessDeniedException` is thrown. If the object does not exist or has been deleted, a `LivelinkObjectNotFoundException` is thrown.

```
ContextId cid = ...
LivelinkId llid1 = new LivelinkId(123);
LivelinkObjectHandler lloh =
      LivelinkRuntime.getInstance().getObjectHandler("livelink");
try {
      LivelinkObject llobj = lloh.get(cid, llid1);
      System.out.println("current object="+llobj);
      LivelinkId llid2 = new LivelinkId(0,123,2);
      llobj = lloh.get(cid,llid2);
      System.out.println("object version 2="+llobj);
}
catch (LivelinkAccessDeniedException lade) {
      System.out.println("access denied");
}
catch (LivelinkObjectNotFoundException lonfe) {
      System.out.println("object not found");
}
```

To read a Livelink object, you can also use the `getAccessible` method. If the object cannot be read, this method does not throw an exception, but returns `null`.

```
LivelinkObject llobj = lloh.getAccessible(cid, llid1);
if (llobj!=null)
{
      System.out.println("current object="+llobj);
}
```

Livelink objects are kept in a separate object cache which reduces the number of LAPI calls. In the **livelink-systems.xml** file, you can configure parameters for the object cache. The file is located in the directory **{WCM installation directory}\config**. An object is removed from the cache when it has been modified in the Livelink system. However, it is not removed immediately. Cleaning the cache is performed at configurable intervals. To configure the interval, modify the “Check interval” parameter of the Livelink pool accordingly (for more details, refer to the online help of the Admin client).

The `invalidate` method can be used to remove a Livelink object from the cache. This ensures that the current version of the Livelink object is used.

```
lloh.invalidate(llid);
LivelinkObject llobj = lloh.get(cid,llid);
```

To check whether another user changed the Livelink object while you edited it, you can use the `isInvalid` method.

```
LivelinkObject llobj = ...
... do something
if (llobj.isInvalid())
{
```

```
      // read fresh instance
      llobj = lloh.get(cid,llid);
}
```

Navigating Livelink

In order to build a navigation, Livelink objects are usually read on the basis of their storage location. For this purpose, the subordinate objects of a Livelink object are determined.

The following code sample retrieves all objects of a Livelink system that the logged-in user is allowed to access.

```
ContextId cid = ...
LivelinkObjectHandler lloh =
      LivelinkRuntime.getInstance().getObjectHandler(systemName);
LivelinkObject llroot = lloh.getRoot(cid);
LinkedList items = new LinkedList();
items.add(llroot);
while (!items.isEmpty())
{
      LivelinkObject llobj = (LivelinkObject)items.removeFirst();
      System.out.println("Found Livelink object: "+llobj.getName());
      if (llobj.hasChildren())
      {
         for (Iterator i = llobj.getChildrenIds().iterator();
            i.hasNext(); )
         {
            LivelinkId childid = (LivelinkId)i.next();
            LivelinkObject llchild = lloh.getAccessible(cid,childid);
            if (llchild!=null) // ignore objects that cannot be read
            {
               items.addLast(llchild);
            }
         }
      }
}
```

Checking Permissions for a Livelink Object

Use the `hasPermission` method of the `LivelinkObjectHandler` interface to check the permissions for a Livelink object.

The following code sample checks whether the current user is allowed to add a Livelink object.

```
ContextId cid = ...
LivelinkObjectHandler lloh = ...
LivelinkId llparentid = new LivelinkId(123);
if (!lloh.hasPermission(cid,llparentid,LivelinkObject.PERM_CREATENODE))
{
      System.out.println("This user is not allowed to add items in parent
         "+llparentid+"! ");
}
```

For more information on managing permission in Livelink, refer to the Javadoc on the `LivelinkObject` interface (read the information about the constants whose names start with `PERM_`), as well as to the LAPI documentation.

Livelink Object Types

Note: The Livelink Integration Manual contains some examples for working with the different Livelink object types.

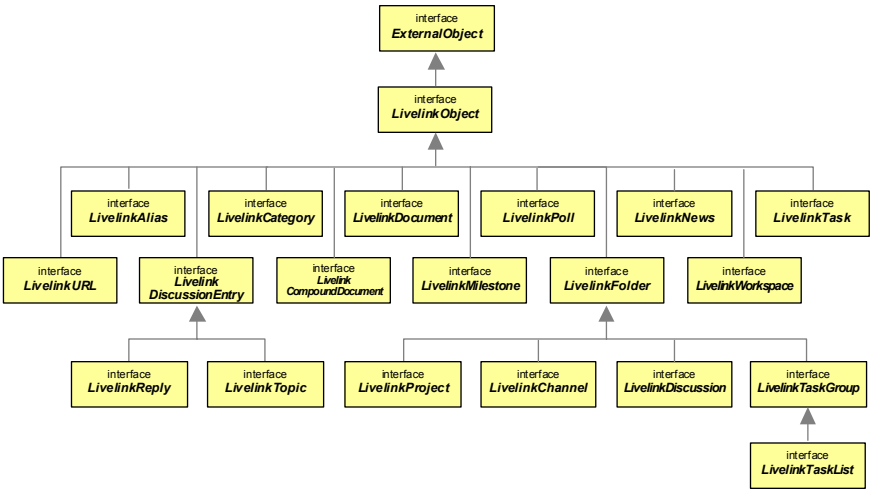


Fig. 13 – Class diagram – Livelink object types

The Livelink object types are represented by constants of the `LivelinkObjectType` interface. The names of these constants start with `SUBTYPE_`. After retrieving a Livelink object, a cast to the appropriate subinterface of `LivelinkObject` is usually required.

```
LivelinkObject llobj = lloh.get(cid, llid);
if (llobj.getObjectType().getSubType()==
      LivelinkObjectType.SUBTYPE_DISCUSSION)
{
      LivelinkDiscussion lldisc = (LivelinkDiscussion)llobj;
      System.out.println("topics="+lldisc.getTopics());
}
```

Retrieving a Livelink Object's Metadata

You can use the `getAttribute` method of the `LivelinkObject` interface to retrieve a Livelink object's metadata. A distinction is made between standard attributes, i.e. attributes which are always present for the respective object type, and attributes from a Livelink category.

The standard attributes are defined by means of the constants of the respective interface of the Livelink object type. The names of these constants start with `META_`. Use the `getAttributeNames` method to retrieve the standard attributes.

The following code sample retrieves all standard attributes of a Livelink object and outputs the associated attribute definitions. If the attribute is represented by a string, the attribute value is output.

```
LivelinkObject llobj = ...
for (Iterator i = llobj.getAttributeNames().iterator(); i.hasNext(); )
{
      Key attr = (Key)i.next();
      AttributeDefinition lldef = llobj.getAttributeDefinition(attr);
      System.out.println("attr="+attr+" attrdef="+lldef);
      Value val = llobj.getAttribute(attr);
      // StringValue ?
      if (lldef.getAttributeType().getId()==AttributeType.TYPEID_STRING)
      {
         StringValue sv = (StringValue)val;
         System.out.println("String attribute value="+sv);
      }
}
```

In the same way, attributes from Livelink categories can be retrieved; you only have to supply the category name.

The following code sample retrieves all Livelink categories of an object and outputs the according attribute values.

```
LivelinkObject llobj = ...
for (Iterator i = llobj.getCategoryNames().iterator(); i.hasNext(); )
{
      String catname = (String)i.next();
      for (Iterator j = llobj.getAttributeNames(catname).iterator();
         j.hasNext(); )
      {
      Key attr = (Key)j.next();
      Value val = llobj.getAttribute(catname, attr);
      System.out.println("category:"+catname+" attribute:"+attr+"
         value:"+val);
      }
}
```

Reading Log Entries

Use the `getAuditTrail` method of the `LivelinkObject` interface to retrieve the history of a Livelink object.

The following code sample reads the log entries of a Livelink object.

```
for (Iterator i = llobj.getAuditTrail().iterator(); i.hasNext(); )
{
      AuditEntry ae = (AuditEntry)i.next();
      System.out.println("date="+ae.getDate()+"
         description="+ae.getDescription()+ "
         event="+ae.getEvent()+" identifier="+ae.getIdentifier()+ "
         user="+ae.getUser().getName());
}
```

Reading Content

Use the `getContent` method of the `LivelinkObject` interface to read the content of a Livelink object. First, you should call the `hasContent` method to check whether the object has any content.

The following code sample reads the content of the current version of a Livelink object.

```
ContextId cid = ...
LivelinkObjectHandler lloh = ...
LivelinkId llid = ...
LivelinkObject llobj = lloh.get(cid, llid);
if (llobj.hasContent())
{
      FileOutputStream fos =
         new FileOutputStream(new File("content.out"));
      try { llobj.getContent(cid,fos); } finally { fos.close(); };
}
```

Former versions of a Livelink object are represented by the `LivelinkVersion` interface. Versions are only provided for Livelink documents (`LivelinkDocument` interface) and Livelink categories (`LivelinkCategory` interface). Only Livelink document versions have content.

The following code sample reads the content of all versions of a Livelink document.

```
ContextId cid = ...
LivelinkObjectHandler lloh = ...
LivelinkId llid = ... // must be a reference to a LivelinkDocument !
LivelinkDocument lldoc = (LivelinkDocument)lloh.get(cid, llid);
for (Iterator i = lldoc.getVersions().iterator(); i.hasNext(); )
{
      LivelinkVersion llversion = (LivelinkVersion)i.next();
      FileOutputStream fos = new FileOutputStream(new File("version_"+
      llversion.getVersionNumber()+".out"));
      try { llversion.getContent(cid,fos); } finally { fos.close(); }
}
```

Performing Actions on a Livelink Object

Livelink objects can be modified via the `LivelinkObjectHandler` interface. You can perform the following actions on a Livelink object: adding, adding content, modifying metadata, reserve, unreserve, and delete.

Adding a Livelink object

All Livelink object types are represented by interfaces. The `LivelinkObjectFactory` class is provided for adding new items. For each Livelink object type, this class provides methods for creating an instance of the appropriate interface. After this, the object is added to Livelink by means of the `LivelinkObjectHandler`.

```
ContextId cid = ...
LivelinkObjectHandler lloh =
      LivelinkRuntime.getInstance().getObjectHandler(systemName);
LivelinkId parentllid = new LivelinkId(123);
LivelinkDocument lldoc =
LivelinkObjectFactory.createDocumentInstance("Demo Document",
      "API test");
lldoc = (LivelinkDocument)lloh.create(cid, parentllid, lldoc);
```

Adding content to a Livelink object

The following code sample adds a file as content to a Livelink document.

```
File file = ...
lldoc = (LivelinkDocument)lloh.addVersion(cid, lldoc.getLivelinkId(),
      file);
```

Changing a Livelink object's metadata

In order to change a Livelink object's metadata, a changeable copy of the Livelink object is created first. In this copy, the new values are set by means of the `setAttribute` method. After this, the changes are saved with the `change` method of the `LivelinkObjectHandler`.

```
LivelinkChangeableObject llchange = lldoc.getChangeable();
llchange.setAttribute(LivelinkObject.META_COMMENT,
      new StringValue("a new comment"));
lloh.change(cid, llchange);
```

Reserving and unreserving Livelink objects

In Livelink, users can reserve objects. The WCM Java API provides the methods `reserve` and `unreserve` (`LivelinkObjectHandler` interface) for this purpose.

```
lloh.reserve(cid, lldoc.getLivelinkId(), null);
// object cannot be modified now
lloh.unreserve(cid,lldoc.getLivelinkId());
// object can be modified now
```

Deleting Livelink objects

Use the `delete` method of the `LivelinkObjectHandler` interface to remove Livelink objects from the Livelink system.

```
lloh.delete(cid, lldoc.getLivelinkId());
```

Relators

You can add objects from external systems as WCM objects to WCM-managed websites. These objects are called *relators*. Livelink is an example for such an external system. This section deals with the following topics:

- the technical concept of external objects, see the following section
- setting up a connection to a Livelink object, see section “Connection to the Livelink Object” on page 197
- adding relators, see section “Adding Relators” on page 199
- modifying relators, see section “Editing Relators” on page 200
- deleting relators, see section “Editing the content of relators” on page 202

External Objects

The `ExternalObject` interface enables Livelink WCM Server to access external systems, e.g. Livelink. An external object is represented by the `ExternalObject` interface. Livelink objects extend this interface (see figure “Class diagram – Livelink object types” on page 190). A relator is a WCM object with an external object as a metadata item.

The following code sample retrieves the external object.

```
ContextId cid = ...
ObjectHandler oh = VipRuntime.getObjectHandler("mywebsite");
ObjectId myrelatorOID = new ObjectId("123");
ObjectData od = oh.get(cid,myrelatorOID);
if (od.isExternalObject())
{
      ExternalObject extobj = od.getExternalObject();
```

```
      // or using an attribute
      extobj = (ExternalObject)
         od.getAttribute(FieldNames.EXTERNALOBJECT,null);
}
```

If Livelink is the external system, external objects are Livelink documents or Livelink folders. The following code sample shows how to convert the instances of `ExternalObject` to the corresponding Livelink interface.

```
ExternalObject extobj = od.getExternalObject();
if (extobj.getType()==ExternalObject.TYPE_LIVELINK)
{
      LivelinkObject llobj = (LivelinkObject)extobj;
      if (llobj.getObjectType().getSubType()==
         LivelinkObjectType.SUBTYPE_DOCUMENT)
      {
         LivelinkDocument lldoc = (LivelinkDocument)llobj;
         ...
      }
      else
      {
         LivelinkFolder llfolder = (LivelinkFolder)llobj;
         ...
      }
}
```

Connection to the Livelink Object

A relator is a WCM object that refers to an external object and is associated with this object. To connect a WCM object to a Livelink object, a Livelink ID and the name of the Livelink system are required.

The basic class for the connection is `ExternalObjectRelator`. The `LivelinkRelator` class extends this basic class and establishes the connection to a Livelink object.

Establishing the connection

The connection can be established to the current version of the Livelink object. In this case, the Edit view of the relator is updated automatically when the Livelink object changes. Alternately, the connection can be established to a fixed version of the Livelink object. In this case, the relator is not updated automatically. The constants `SYNCHMODE_RELEASE_FIXED` and `SYNCHMODE_ALWAYS_FIXED` of the `ExternalObject` interface determine the connection type.

The following example illustrates how to set up a connection to the current version of the Livelink object.

```
// link to the current version of node ID 123 in system livelink
// and update relator if Livelink object changes within Livelink
LivelinkRelator llrel = new LivelinkRelator("livelink",
      new LivelinkId(123), ExternalObject.SYNCMODE_RELEASE_FIXED);
```

The following code sample establishes a connection to a fixed version of a Livelink object.

```
// link to version 2 of node ID 123 in system livelink
// and always use this version
LivelinkRelator llrel = new LivelinkRelator("livelink",
      new LivelinkId(0,123,2), ExternalObject.SYNCMODE_ALWAYS_FIXED);
```

The connection information is required for adding relators. For existing relators, this information can be retrieved from the `ObjectData` instance.

```
ObjectData od = ...
if (od.getObjectType().isLivelinkRelator())
{
      LivelinkRelator llrel =
         (LivelinkRelator)od.getExternalObjectRelator();
```

```
      // or using an attribute
      llrel = (LivelinkRelator)od.getAttribute
         (FieldNames.EXTERNALOBJECTRELATOR,null);
}
```

Changing the connection

The connection properties of a relator can be modified subsequently. The following example illustrates how to refer to a different version of the Livelink object.

```
ContextId cid = ...
ObjectHandler oh = ...
ObjectData od = ...
// update link to Livelink
LivelinkRelator llrel = (LivelinkRelator)od.getExternalObjectRelator();
LivelinkId llid = llrel.getLivelinkId();
LivelinkId llnewid =
      new LivelinkId(llid.getNodeId(),llid.getVersion()+1);
llrel.setLivelinkId(llnewid);
// change metadata
od.setExternalObjectRelator(llrel);
oh.change(cid,od,"relator test");
```

Adding Relators

The first step of adding a relator is to create an `ObjectData` object of type “Livelink relator” or “Livelink folder relator”.

```
ContextId cid = ...
ObjectHandler oh = ...
ObjectId parentOID = new ObjectId("1");
ObjectId templateOID = null;
ObjectType oType =
      oh.getWebsite().getObjectType(ObjectType.LIVELINKRELATOR);
ObjectData od = oh.createInitialObjectData(parentOID, templateOID,
      oType, "myrelator");
```

After this, a connection between the WCM object and the Livelink object is established and set in the `ObjectData` instance.

```
LivelinkRelator llrel = new LivelinkRelator("livelink",
      new LivelinkId(123), ExternalObject.SYNCMODE_RELEASE_FIXED);
od.setExternalObjectRelator(llrel);
```

Finally, the WCM object is created.

```
ObjectId newrelatorOID = oh.create(cid,od,null, "relator test");
```

In the example above, `null` is passed in the file argument as relators do not have content of their own in the WCM system. The content of the relator is stored in the Livelink database.

Editing Relators

The WCM Java API provides functions for editing the metadata and content of relators.

Modifying Livelink metadata

Each Livelink object has a number of metadata. Thus, each relator has the Livelink metadata of the associated Livelink object in addition to the WCM metadata. WCM metadata and Livelink metadata can be modified simultaneously.

The following code sample changes the relator's title and the comment of the associated Livelink object, i.e. the modifications refer both to the WCM system and to the Livelink system.

```
ContextId cid = ...
ObjectHandler oh = ...
ObjectData od = ...
// change title
```

```
od.setTitle("A new title");
// change Livelink comment
LivelinkObject llobj = (LivelinkObject)od.getExternalObject();
LivelinkChangeableObject llchange = llobj.getChangeable();
llchange.setAttribute(LivelinkObject.META_COMMENT,
      new StringValue("A new Livelink comment"));
// pass change to ObjectData instance
od.setExternalObject(llchange);
// change it now !
oh.change(cid,od,"relator test");
```

Editing deployment-specific metadata

In Livelink, any content can be added to a document as a new version. It is possible, for example, that the first version of a document is a GIF image and the second version an HTML file. This affects the deployment of relators as the relator's representation in the file system may change. For this reason, a distinction is made between the “normal” WCM object type and the object type used for the deployment of relators.

```
ContextId cid = ...
ObjectHandler oh = ...
ObjectData od = ...
// output object type for WCM object
System.out.println("object type="+ od.getObjectType().getName());
// output object type used by the deployment
System.out.println("deployment object type="+
      od.getDeploymentObjectType().getName());
```

This difference between object type and deployment object type only exists for relators. The icon representing the object type in the Content client and the MIME type of relators is not determined by the object type, but by the WCM object itself. For this reason, the icon's URL and the MIME type can be retrieved from an `ObjectData` instance.

```
String imageurl = od.getImageURL("intranet_edit");
String mimetype = od.getMimeType();
System.out.println("objectdata: imageurl="+imageurl+"
mimetype="+mimetype);
// returns different results for relators
imageurl = od.getObjectType().getImageURL("intranet_edit");
mimetype = od.getObjectType().getMimeType();
System.out.println("objecttype: imageurl="+imageurl+"
mimetype="+mimetype);
```

Editing the content of relators

The content of relators is stored in the Livelink database. Nevertheless, it can be edited via the WCM Java API – just like the content of other WCM objects (see section “Editing Content” on page 126). In order to decide whether the content can be changed and how to analyze it, the relator's deployment object type is used.

The following code sample reads the content of the relator and copies it to a new file.

```
ContextId cid = ...
ObjectHandler oh = ...
ObjectId relatorOID=new ObjectId("123");
// checkout
oh.checkOut(cid,relatorOID,"test relator");
// get content
ObjectData od = oh.get(cid,relatorOID);
File outputFile = od.getCheckOutContent(cid,"intranet_edit", null);
... modify content
// check in
oh.checkIn(cid, relatorOID, outputFile, "test relator");
```

Deleting Relators

Relators are WCM objects. For this reason, they can be deleted via the `ObjectHandler` interface. Please note the following: In the connection properties of a relator, you can determine whether the referenced Livelink object is to be deleted when the relator is deleted. To set this property, use the `setLiable` method of the `ExternalObjectRelator` class. If this method is not called, the Livelink object will not be deleted by default.

The following code sample sets this property and deletes the Livelink object together with the relator.

```
LivelinkRelator llrel = new LivelinkRelator("livelink",
      new LivelinkId(123), ExternalObject.SYNCMODE_RELEASE_FIXED);
// delete Livelink object when WCM object is deleted
llrel.setLiable(ExternalObject.LIABLE_VIP);
od.setExternalObjectRelator(llrel);
oh.change(cid,od,null);
// destroy both objects
List oidlist = new ArrayList();
oidlist.add(od.getObjectId());
oh.destroy(cid, oidlist, null, null);
```

CHAPTER 8

Application Examples

This chapter introduces several simple application examples designed to show how server agents can be developed.

- Section “Basic Class: `ExampleAgent`” on page 207 introduces an abstract agent class that already implements the necessary properties and demonstrates the localization of texts.
- The agent “AutoSubmissionAgent” is an active agent. When a website is ready for operation, this agent submits all WCM objects of the website if they have not yet been released. This example shows how to register for run level events.

  See section “Automatic Submission” on page 209
- The agent “VetoChangeTitleAgent” demonstrates the veto mechanism. The users of a certain group are not allowed to change the title of WCM objects. They may only use a predefined template. The agent checks whether these rules are adhered to and, if necessary, vetoes the change of a title or template.

  See section “Vetoing Metadata Changes” on page 212

- The agent “DplEventTester” illustrates registration for deployment events.

  See section “Monitoring Deployment Events” on page 215

- The agent “KeywordAnalyzerAgent” searches keywords, target group, and description of a newly created or changed WCM object for German umlauts and some special characters and converts them by means of the WCM Java API (“ä” into “ae”, “ö” into “oe”, etc.).

  See section “Converting Special Characters” on page 221

- The agent “TranslationJobAgent” works in multi-language websites. As soon as an object to be translated is added to the website, the agent informs the editors responsible for translation.

  See section “Multi-Language Websites” on page 228

The examples shown here are primarily designed to demonstrate how server agents can be programmed. For the sake of simplicity, certain functions were not implemented at all or only partially. This applies, for example, to the areas of error handling and error logging. Under real conditions, these functions must be implemented using the corresponding interfaces.

Note: The JAVA files of the server agents are located in the directory **\examples\serveragents** of the WCM installation directory.

Basic Class: ExampleAgent

This example implements a basic class from which the following example agents derive. In this basic class, the common properties of the ServerAgent interface (manufacturer, description, and version of the agent, as well as the required Livelink WCM Server version) are implemented. The UserMessage class can be used to provide a localized description of the agent.

Configuration parameters

As the ExampleAgent is a basic class, no parameters are expected.

```
// Abstract basic class for deriving agents (implements methods
// from de.gauss.vip.api.ServerAgent that return constant values which
// are set via a constructor).
public abstract class ExampleAgent
    implements ServerAgent
{
    // final variables for holding constant values
    private final UserMessage msgM;
    private final String manufacturerM;
    private final String versionM;
    private final int majorM;
    private final int minorM;

    // the only constructor
    public ExampleAgent(String manufacturerArg, String msgKey,
                        String versionArg, int majorArg,
                        int minorArg)
    {
        manufacturerM = manufacturerArg;
        msgM          = new UserMessage(msgKey);
        versionM      = versionArg;
        majorM        = majorArg;
        minorM        = minorArg;
    }

    /**
     * de.gauss.vip.api.ServerAgent implementation (except start(...)
     * and stop())
     */
    public String getManufacturer()
    {
```

```
        return manufacturerM;
    }

    public String getDescription(Locale locale)
    {
        //The message keys must be defined in the
        // "UserMessage_<langcode>.properties" file which
        // is loaded via a class loader.
        return msgM.getString(locale);
    }

    public String getVersion()
    {
        return versionM;
    }

    public int getRequiredVIPMajorVersion()
    {
        return majorM;
    }

    public int getRequiredVIPMinorVersion()
    {
        return minorM;
    }
}
```

Automatic Submission

This agent demonstrates how active agents are programmed. The agent is not controlled by staging events, it becomes active directly after the configured website has been started.

The WCM objects of the website cannot be accessed until the website has reached the run level `WEBSITE_UP`. For this reason, the agent registers for the run level change event and checks whether the website run level has been reached. Only then does it become active.

After the website has booted, the agent looks for all WCM objects that have never been released (for this purpose, it uses the search function from the `ObjectHandler` interface) and submits the WCM objects it finds.

Configuration parameters

- `website`: name of the website to be monitored. The parameter is of type `String`.
- `agent-user` and `agent-password`: user ID and password of a user with read and write permissions for the monitored website. Both parameters are of type `String`. The agent logs the specified user in to the system at startup because the context ID of a logged-in user is later required for submitting objects at a later stage.

```
// This agent registers itself for RUNLEVEL_IS events for a
// configurable website and synchronously submits any edited object
//that has never been released.
public class AutoSubmissionAgent
      extends ExampleAgent
      implements EventListener
{
      //holds EventDispatcher (singleton)
      private final static EventDispatcher eventDispatcherM
         = VipRuntime.getEventDispatcher();

      //set from "website" parameter
      private final String websiteNameM;
      //set from "agent-user" parameter
      private final String agentUserM;
```

```
//set from "agent-password" parameter
private final String agentPasswordM;
 private Filter filterM;

//Constructor reads configuration parameters and prepares the
//filter.
public AutoSubmissionAgent(Properties config)
   throws VipApiException
{
   //"AUTOSUBMISSIONAGENT_DESC" will be localized via
   //"UserMessage_<langcode>.properties" file
   super("Company.Example", "AUTOSUBMISSIONAGENT_DESC",
                                                    "1.1", 8, 2);
   websiteNameM = config.getProperty("website");
   agentUserM     = config.getProperty("agent-user");
   agentPasswordM = config.getProperty("agent-password");

}

/**
 * de.gauss.vip.api.ServerAgent implementation that is not
 * covered by ExampleAgent
 */

//register this instance for events
public boolean start(String serverType, int majorVersion,
                             int minorVersion, String patchLevel)
{
   eventDispatcherM.addListener(null, Event.RUNLEVEL_IS, this);
   return true; //success
}

//deregister this instance for all websites/events
public void stop()
{
   eventDispatcherM.removeListener(this);
}

/**
 * de.gauss.vip.api.event.EventListener implementation
 */

//If the configured website has just been started, this method
//submits any edited object that has never been submitted to QA
//(the submisson is done synchronously --- so event propagation
//will pause until the submission completes). For 'real' agents
//this should be changed to asynchronous submission, but special
//care would have to be taken for context refresh and subsequent
//events, e.g. WEBSITE_INACCESSIBLE).
```

```
public void performVipEvent(Event event)
{
   Integer level  = (Integer)event.getArgument(Event.ARG_NEW);
   String website = (String)event.getArgument(Event.ARG_WEBSITE);

   if (level.intValue() == Runlevel.WEBSITE_UP
      && website != null && website.equals(websiteNameM))
   {
         ContextHandler cxtHandler = null;
         ContextId contextId       = null;
      try
      {
         //construct filter
         ObjectState objState =
               ObjectHandlerUtil.getObjectState(websiteNameM,
                                                ObjectState.EDITED);
         Filter objStateFilter =
               new EqualFilter(SearchableKeys.STATE,objState);
         Filter versionFilter  =
               new AndFilter(new EqualFilter
                                (SearchableKeys.VERSION_MINOR,
                           new LongValue(0)),
                           new EqualFilter(SearchableKeys.
                                VERSION_MAJOR,
                           new LongValue(0)));

         filterM = new AndFilter(objStateFilter, versionFilter);
         ObjectHandler objHandler
                     = VipRuntime.getObjectHandler(websiteNameM);
         cxtHandler = VipRuntime.getContextHandler();
         contextId  = cxtHandler.login(agentUserM,
                                agentPasswordM.toCharArray());

         //find objects and sort the result so that parents
         //will be displayed before their children
         List objects = objHandler.sortParentsFirst(contextId,
                     objHandler.filter(contextId, filterM, null,
                                                  null, 0, -1));
         //submit (no pending release date and no e-mail)
         objHandler.submit(contextId, objects, null,
                                "agent's remark", null);
      }
      catch (VipApiException vae)
      {
         vae.printStackTrace();
      }
      finally
      {
         try
```

```
                {
                   if (cxtHandler != null && contextId != null)
                            cxtHandler.logout(contextId);
                }
                catch (VipApiException vax)
                {
                   vax.printStackTrace();
                }
             }
          }
       }
}
```

Vetoing Metadata Changes

This agent demonstrates how to register for and process preparation events (`PrepareEvents`). The agent prohibits WCM object titles from being changed or using a template other than the configured one. If the title has been changed or a template has been selected that does not correspond to the configured template of the agent, the agent throws a `VetoException`. This prevents the action from being performed. The localized message used will then be displayed, e.g. in the user's Content client.

This example uses an inner class to implement the `PrepareEventListener` interface.

Configuration parameters

- `website`: name of the website to be monitored. The parameter is of type `String`.
- `template`: the OID of the template to be used. The parameter is of type `String`.

```
// This agent registers an inner class instance for
// PrepareEvent.PREPARE_OBJECT_CHANGE_METADATA events for a
```

```
// configurable website and prevents title changes and template
// assignments via veto exceptions.
public class VetoChangeTitleAgent
      extends ExampleAgent
{
      //These keys will be localized via
      //"UserMessage_<langcode>.properties" file
      private final static String templateVetoKeyM
         = "VETO_TEMPLATE_FOR_{0}_SHOULD_BE_{1}";

      private final static String titleVetoKeyM
         = "VETO_TITLE_FOR_{0}_NOT_CHANGEABLE";

      //keep the EventDispatcher singleton
      private final static EventDispatcher eventDispatcherM
         = VipRuntime.getEventDispatcher();

      //blank final fields

      //set from "website" config parameter
      private final String websiteNameM;
      //set from "template" config parameter
      private final ObjectId templateM;

      //Construct the PrepareChangeListener (inner class; c.f. below)
      //that will receive events after it has been registered (from
      //within start(...))
      private final PrepareEventListener eventListenerM
         = new PrepareChangeListener();

      //constructor reads configuration parameters
      public VetoChangeTitleAgent(Properties config)
      {
         super("Company.Example", "VETOCHANGETITLEAGENT_DESC",
                                              "1.1", 8, 2);
         websiteNameM = config.getProperty("website");
         templateM    = new ObjectId(config.getProperty("template"));
      }

      /**
       * de.gauss.vip.api.ServerAgent implementation that is not covered
       * by ExampleAgent.
       */

      //register the PrepareChangeListener instance for website/event
      public boolean start(String serverType, int majorVersion,
                           int minorVersion, String patchLevel)
      {
         eventDispatcherM.addListener(websiteNameM,
```

```
                        PrepareEvent.PREPARE_OBJECT_CHANGE_METADATA,
                        eventListenerM);
      return true; //this agent wants to be stopped explicitly
   }

   //deregister the PrepareChangeListener instance for all
   //websites/events
   public void stop()
   {
      eventDispatcherM.removeListener(eventListenerM);
   }

   //will receive events and accesses some
   //fields from enclosing class/instance
   private class PrepareChangeListener
      implements PrepareEventListener
   {
      //throw veto exception, if wrong template assignment or if
      //title has been changed
      public void performVipPrepareEvent(PrepareEvent event)
         throws VetoException
      {
         ObjectId objId
            = (ObjectId)event.getArgument(Event.ARG_OID);
         RepositoryEntry changed
            = (RepositoryEntry)event.getArgument(Event.ARG_NEW);

         if (changed.containsKey(FieldNames.TEMPLATE))
         {
            ObjectId newTemplate
               = (ObjectId)changed.get(FieldNames.TEMPLATE);
      if (newTemplate != null &&
                  !newTemplate.equals(templateM))
               throw new VetoException(new
                        UserMessage(templateVetoKeyM,objId.getId(),
                                             templateM.getId()));
         }
         if (changed.containsKey(FieldNames.TITLE))
            throw new VetoException(new UserMessage(titleVetoKeyM,
                                                objId.getId()));
      }
   }
}
```

Monitoring Deployment Events

This example shows a simple implementation of an agent that monitors all deployment events and writes them to the console.

Configuration parameters

No parameters are required for this agent.

```
/**
 * Title:
 * Description:  A small agent for deployment events.<br>
 *               Starts a thread printing out deployment events.
 * Copyright:    Copyright (c) 2002
 * Company:
 * @author
 * @version 1.0
 */
public class DplEventTester
      extends ExampleAgent
      implements DeploymentEventListener, ServerAgent, Runnable
{
      private static int counterM = 0;
      private Object mutexM = new Object();
      private String nameM = "DeplEventTester_";
      private boolean stopM = false;
      private LinkedList todoM;
      private de.gauss.vip.config.PropertyMap pmM;

      public DplEventTester(de.gauss.vip.config.PropertyMap pm)
      {
         super("Gauss Interprise AG", "Java API Test Deployment Agent",
                                                          "0.2", 8, 0);
         pmM = pm;
         counterM++;
         nameM = nameM + counterM;
         todoM = new LinkedList();
      }

      //--- interface ServerAgent. Caution: some methods are inherited
      //from the super class.

      /**
       * Starts a thread for printing out events.
       *
       * @param serverType the type of the WCM server that will
```

```
 * execute this agent.
 * @param majorVersion the major release number of
 * Livelink WCM Server.
 * @param manorVersion the minor release number of
 * Livelink WCM Server.
 * @param patchLevel the micro (patch level) release number of
 * Livelink WCM Server.
 * @return <tt>true</tt> because the agent wants {@link #stop()} to
 * be called.
 */
public boolean start(String serverType, int majorVersion,
                     int minorVersion, String patchLevel)
{
   try
   {
      stopM = false;
      Thread thread = new Thread(this, nameM);
      //Start the thread's run method.
      thread.start();
      return true;
   }
   catch (Exception ex)
   {
      ex.printStackTrace();
      return false;
   }
}

/**
 * Stops the thread that the agent has started.
 */
public void stop()
{
   synchronized(mutexM)
   {
      stopM = true;
      mutexM.notify();
   }
}

//--- interface EventListener
/**
 * Adds the given event to the internal todo queue.
 *
 * @param event Event to be handled.
 */
public void performVipDeploymentEvent(DeploymentEvent event)
{
   synchronized(mutexM)
```

```
   {
      todoM.addLast(event);
      mutexM.notify();
   }
}

//--- interface Runnable
/**
 * Registers for deployment events.
 * Waits for events to be performed (in a loop).
 * Leaves the loop as soon as {@link #stop()} has been invoked.
 * Deregisters for deployment events before returning.
 */
public  void run()
{
   DeploymentEvent event = null;

   try
   {
      System.out.println( "Thread " + nameM
               + " has configuration " + reportPM(pmM));
      VipRuntime.getDeploymentEventDispatcher().addListener(
               DeploymentEvent.FILE_CHANGED, this);
      VipRuntime.getDeploymentEventDispatcher().addListener(
               DeploymentEvent.FILE_DELETED, this);
      VipRuntime.getDeploymentEventDispatcher().addListener(
               DeploymentEvent.DEPLOYMENT_SYSTEM_CREATED, this);
      VipRuntime.getDeploymentEventDispatcher().addListener(
               DeploymentEvent.DEPLOYMENT_SYSTEM_DELETED, this);
   }
   catch (Exception ex)
   {
      ex.printStackTrace();
   }
   do
   {
      try
      {
         synchronized(mutexM)
         {
            if (!stopM)
            {
               if (todoM.isEmpty())
               {
                  try
                  {
                     mutexM.wait(300000);
                  }
                  catch(InterruptedException ex)
```

```
                    {
                       //Don't do anything!
                    }
                 }
                 else
                 {
                    event = (DeploymentEvent)todoM.removeFirst();
                 }
              }
           }
           if (event != null)
              printEvent(event);
           event = null;
        }
        catch (Exception ex)
        {
           ex.printStackTrace();
        }
     } while(!stopM);
     try
     {
        VipRuntime.getDeploymentEventDispatcher().removeListener(
                  DeploymentEvent.FILE_CHANGED, this);
        VipRuntime.getDeploymentEventDispatcher().removeListener(
                  DeploymentEvent.FILE_DELETED, this);
        VipRuntime.getDeploymentEventDispatcher().removeListener(
                  DeploymentEvent.DEPLOYMENT_SYSTEM_CREATED, this);
        VipRuntime.getDeploymentEventDispatcher().removeListener(
                  DeploymentEvent.DEPLOYMENT_SYSTEM_DELETED, this);

        System.out.println( "Thread " + nameM
                  + " will be stopped immediately.");
     }
     catch (Exception ex)
     {
        ex.printStackTrace();
     }
  }

  //Tools
  /**
   * Prints the given event.
   * @param event Event to be printed.
   */
  private void printEvent(DeploymentEvent event)
  {
     switch (event.getId())
     {
        case DeploymentEvent.FILE_CHANGED:
```

```
            case DeploymentEvent.FILE_DELETED:
               System.out.println(
                  "Thread   " + nameM + "\n" +
                  "* Event: " + event.getId()+
                  "("+reportEvent(event.getId())+")\n"+
                  "* Dpl:   " + (StringValue)event.getConstraint() +
                  "\n" + "* Oid:   " +
                  (ObjectId)event.getArgument(DeploymentEvent.ARG_OID)+
                  "\n" +
                  "* Path:  " + (StringValue)event.getArgument(
                                    DeploymentEvent.ARG_PATH)+"\n" +
                  "* Url :  " + (StringValue)event.getArgument(
                                       DeploymentEvent.ARG_URL));
               break;
            case DeploymentEvent.DEPLOYMENT_SYSTEM_CREATED:
            case DeploymentEvent.DEPLOYMENT_SYSTEM_DELETED:
               System.out.println(
                  "Thread   " + nameM + "\n" +
                  "* Event: " + event.getId()+
                  "("+reportEvent(event.getId())+")\n"+
                  "* Dpl:   " + (StringValue)event.getArgument(
                              DeploymentEvent.ARG_DEPLOYMENT_SYSTEM));
               break;
            default:
               System.out.println(
                  "Thread   " + nameM + "\n" +
                  "* Event: " + event.getId()+
                  "("+reportEvent(event.getId())+")");
               break;
         }
      }

      /**
       * Converts the given event ID to a <tt>String</tt>.
       * @param id Event id.
       * @return String describing the event.
       */
      private String reportEvent(int id)
      {
         switch (id)
         {
            case DeploymentEvent.FILE_CHANGED:
               return "FILE_CHANGED";
            case DeploymentEvent.FILE_DELETED:
               return "FILE_DELETED";
            case DeploymentEvent.DEPLOYMENT_SYSTEM_CREATED:
               return "DEPLOYMENT_SYSTEM_CREATED";
            case DeploymentEvent.DEPLOYMENT_SYSTEM_DELETED:
               return "DEPLOYMENT_SYSTEM_DELETED";
```

```
            default:
                return "unknown";
        }
    }

    /**
     * Converts the given property map to a <tt>String</tt>. Traverses
     * the PropertyMap using a recursion if neccessary.
     * @param pm Property map containing configuration data.
     * @return String describing the contents of the given property
     * map.
     */
    private String reportPM( de.gauss.vip.config.PropertyMap pm )
    {
        if ( pm!= null )
        {
            List childKeys = pm.getChildrenKeys();
            if ( childKeys != null )
            {
                StringBuffer ret = new StringBuffer();
                Iterator it = childKeys.iterator();
                while( it.hasNext() )
                {
                    String key = (String)it.next();
                    ret = ret.append(key);
                    ret = ret.append(":");
                    try
                    {
                        if ( pm.hasPropertyMap( key) )
                        {
                            ret.append(
                                reportPM( pm.getPropertyMap( key )));
                        }
                        else
                        {
                            String value = pm.getPropertyValue( key );
                            ret = ret.append( value );
                            ret = ret.append(" ");
                        }
                    }
                    catch ( KeyNotFoundException notFound )
                    {
                        ret = ret.append("error! ");
                    }
                }
```

```
                return ret.toString();
            }
        }
        return "-";
    }
}
```

Converting Special Characters

This example shows an agent that searches the metadata “Keywords”, “Description”, and “Target group” of a newly created or changed WCM object for German umlauts or some special characters and converts them by means of the WCM Java API (“ä” to “ae”, “ö” to “oe”, etc.).

For this purpose, the agent registers an inner class `StringAnalyzer` for the events `OBJECT_CREATED` and `OBJECT_METADATA_CHANGED`. Moreover, the agent logs in a configurable user to the system via the WCM Java API because the context ID of a logged-in user is required for changing the metadata of a WCM object. This user must have the right “Change metadata”, which must be defined in the access control list for the respective WCM object.

If these two events occur, the `StringAnalyzer` checks whether the metadata “Keywords”, “Description”, and “Target” group contain terms with German umlauts or special characters and converts these characters accordingly. Afterwards, the `StringAnalyzer` causes a metadata change via the WCM Java API. For this purpose, the `change` method is called for the object whose metadata were changed.

The agent must first be configured with the user ID and password in the Admin client (via the parameters “username” and “password”). Also, at least one website must be specified that the agent is to monitor. For this purpose, the “websites” parameter must be created for the agent in the Admin client. If several websites are to be monitored, they must be separated by semicolons. The constructor reads the parameters “website”, “username”, and “password” from the configuration data that was specified for the agent in the Admin client and saves them in the respective variables of the agent.

From the file **UserMessage_{language code}.properties**, which is located in the directory **{WCM installation directory}\config\resource**, the text is read that is to be written to the log of the object after the metadata change (key: “KEYWORD_AGENT_MESSAGE”).

Also, an instance of the `StringAnalyzer` class is created, which will later be used for analyzing and, if necessary, changing the metadata “Keywords”, “Description”, and “Target group”.

Configuration parameters

- `websites`: list of the names of the websites to be monitored separated by semicolons. The parameter is of type `String`.
- `username` and `password`: user ID and password of a user with read and write permissions for all monitored websites. Both parameters are of type `String`.

```
public class KeywordAnalyzerAgent extends ExampleAgent implements
ServerAgent, EventListener
{
...
      public KeywordAnalyzerAgent(Properties properties)
      {
         super("company.example", "KEYWORDAGENT_DESC", "1.1", 8, 2);
         String w = (String) properties.get("websites");
         if( w != null )
         {
            StringTokenizer websiteNames = new StringTokenizer
                                                   ( w, ";", false );
```

```
            websitesM = new HashSet();
            while ( websiteNames.hasMoreElements())
            {
               String wsName = websiteNames.nextToken();
               websitesM.add(wsName);
            }
         }
         //set user ID and password for login..
         userNameM = (String) properties.getProperty("username");
         String pwd = (String) properties.getProperty("password");
         if( pwd != null )
            userPasswordM = pwd;
         UserMessage uM = new UserMessage("KEYWORD_AGENT_MESSAGE");
         messageM = uM.getString();
         //create StringAnalyzer ..
         analyzerM = new StringAnalyzer();
      }

.....
```

The agent is started by the method `start`. First, it is checked whether the agent was started on a server of type `TYPE_CONTENT` (i.e. a Content server). Only if this is the case, starting is continued. Otherwise, starting is aborted as the agent can only be executed on Content servers.

Afterwards, the agent registers for the `RUNLEVEL_IS` event, which will later be used to determine whether the server has already reached a run level in which a user can be logged in. For this purpose, the `StringAnalyzer` created in the constructor is registered for the events `OBJECT_METADATA_CHANGED` and `OBJECT_CREATED` for all desired websites because analysis and conversion of the metadata to be checked is to be performed for each newly created WCM object and after each change to the metadata of a WCM object.

```
public boolean start(String serverType, int majorVersion, int
minorVersion, String patchLevel)
{
      System.out.println(" Servertype : " +  serverType);
      if( !serverType.equals(Server.TYPE_CONTENT) )
      {
         System.err.println("KeywordAnalyzerAgent: start() Server type
                                                    is not CONTENT!");
         return false;
      }
      if( websitesM == null || websitesM.isEmpty() )
      {
         System.err.println("KeywordAnalyzerAgent: start() No website
                                                          specified!");
         return false;
      }
      if( userNameM == null )
      {
         System.err.println("KeywordAnalyzerAgent: start() No user
                                                          specified!");
         return false;
      }
      Iterator wsIter = websitesM.iterator();
      while( wsIter.hasNext() )
      {
         String w = (String) wsIter.next();
         //register agent on RUNLEVEL_IS, for login purposes..
         VipRuntime.getEventDispatcher().addListener(null,
                                              Event.RUNLEVEL_IS, this);
         //register StringAnalyzer on Events METADATA_CHANGED and
         //OBJECT_CREATED ..
         VipRuntime.getEventDispatcher().addListener
                        (w, Event.OBJECT_METADATA_CHANGED, analyzerM);
         VipRuntime.getEventDispatcher().addListener
                                 (w, Event.OBJECT_CREATED, analyzerM);
      }
      return true;
}
......
```

The agent logs in the specified user in the method `performVipEvent` when the run level required for this action is reached in one of the monitored websites. Logout of the user is performed by the method `stop` of the agent. This is, however, not explained here.

```
public void performVipEvent(Event event)
{
      Integer level  = (Integer)event.getArgument(Event.ARG_NEW);
      String eventWebsite = (String)
                                 event.getArgument(Event.ARG_WEBSITE);
      if (level.intValue() == Runlevel.WEBSITE_UP &&
                        websitesM.contains(eventWebsite) && ! bLoginM )
      {
         try
         {
            ContextHandler ch = VipRuntime.getContextHandler();
            if( userNameM.equalsIgnoreCase("world") )
            {
               ctxId = ch.getWorldContextId();
            }
            else
            {
               ctxId = ch.login(userNameM, userPasswordM);
               //context should not time out.
               ch.startContextRefresh(ctxId);
            }
            bLoginM= true;
         }
         catch ( VipApiException vaex )
         {
         ...........
}
.....
```

The inner class `StringAnalyzer` is responsible for analyzing and, if necessary, changing the metadata to be monitored. The `StringAnalyzer` class is implemented as `EventListener` to make it possible to register objects of this class for certain events. In the `performVipEvent` method, the metadata to be checked are retrieved and, if necessary, modified by calling the `correctString` method of the `StringAnalyzer`.

```
public class StringAnalyzer implements EventListener
{
      public void performVipEvent(Event event)
      {
         try
         {
            String website     = (String) event.getConstraint();
            ObjectHandler oh   = VipRuntime.getObjectHandler(website);
            ObjectId oid       = (ObjectId)
                                       event.getArgument(Event.ARG_OID);
            ObjectData od      = oh.get(ctxId, oid);
            String targetGroup = od.getTargetGroup();
            Set keywords    = od.getKeywords();
            String description = od.getDescription();
            boolean bChange = false;
            if( targetGroup != null && targetGroup.length() > 0 )
            {
               String newTargetGroup = correctString(targetGroup);
               if( !targetGroup.equals(newTargetGroup) )
               {
                  od.setTargetGroup(newTargetGroup);
                  bChange = true;
               }
            }
            if ( keywords != null && ! keywords.isEmpty() )
            {
               HashSet newKw = new HashSet();
               Iterator keyWordIter = keywords.iterator();
               while ( keyWordIter.hasNext())
               {
                  String keyword = ( String )keyWordIter.next();
                  String newKeyword = correctString(keyword);
                  newKw.add(newKeyword);
               }
               od.setKeywords( newKw);
               bChange = true;
            }
            if ( description != null && description.length() > 0 )
            {
               String newDescription = correctString(description);
               if( !description.equals(newDescription) )
               {
                  od.setDescription(newDescription);
                  bChange = true;
               }
            }
            if ( bChange )
```

```
            {
               oh.change(ctxId, od, messageM);
            }
         }
         catch (VipApiException vae)
         {
            System.err.println("KeywordAnalyzerAgent: performVipEvent()
                                                              " + event );
            vae.printStackTrace();
         }
      }
}
```

The `correctString` method modifies the supplied strings by filtering and converting German umlauts and other special characters.

```
public String correctString(String str)
{
      StringBuffer output = new StringBuffer();
      for( int i = 0; i < str.length(); i++ )
      {
         switch(str.charAt(i))
         {
            case ';':
               output.append(',');
               break;
            case 'ä':
               output.append("ae");
               break;
            case 'Ä':
               output.append("Ae");
               break;
            case 'ö':
               output.append("oe");
               break;
......
            default:
               output.append(str.charAt(i));
         }
      }
      return output.toString();
}
```

Multi-Language Websites

This example describes an agent for multi-language websites. In multi-language websites, objects are maintained in different languages. As soon as an object to be translated (source object) is added to the website, the agent informs the editors responsible for translation by e-mail. For this purpose, the attribute set “html” in the monitored website must be extended by the attribute `translate`. This attribute indicates whether an object must be translated. If an editor has finished working on an object that is to be translated, the editor changes the value of this attribute in the metadata of the object to `true`.

For notifying the editor responsible for the translation, the agent registers for the event `OBJECT_METADATA_CHANGED`. For all objects for which it is informed about this event, the agent checks whether the attribute `translate` has the value `true`. If this is the case, the agent creates a new object of type “HTML page” for each supported language by means of the respective API call. After the required objects have been created, the value of the attribute `translate` of the source object is set to `false`.

> **Note:** As the agent is intended for demonstration purposes only and is not to be too complex, the agent creates the new objects directly below the root object of the website. This mechanism would have to be adapted in practice.

The content of the new objects is the content of the source object. After the new objects have been created, the editors responsible for translation are informed by e-mail. The attachment of the e-mail contains the text to be translated.

Configuration parameters

- `username` and `password`: user ID and password of a user used for creating the objects. The user must have read and write permission in the monitored website. Both parameters are of type `String`.

- `translation_lang_user`: parameter for specifying the supported languages and the responsible editors. This parameter is of type `String`.

  Here, the locales of the supported languages followed by the user IDs of the responsible translators must be entered. As separator, a comma without space is used.

  Example: The user “jstein” is responsible for translation into English; the user “claurent” is responsible for translation into French. This results in the following entry:
  `translation_lang_user="en_US,jstein,fr_FR,claurent"`

The constructor of the agent reads the values for the user ID, the password, and the assignment of the editors to the languages from the configuration of the agent.

```
public TranslationJobAgent(Properties properties)
{
      super("company.example", "TRANSLATIONJOB_AGENT_DESC", "1.1",
                                                                  8, 2);
      userNameM   = (String) properties.get("username");
      String pwd = (String) properties.get("password");
      if( pwd != null )
         passwordM = pwd;
      loc_usersM = (String)properties.get("translation_lang_user");
      UserMessage uM = new UserMessage("TRANSLATIONJOB_AGENT_MESSAGE");
      messageM = uM.getString();
}
```

By means of the `start` method, the agent is registered for the events `RUNLEVEL_IS` and `OBJECT_METADATA_CHANGED`. Registration for the event `RUNLEVEL_IS` is required for logging the agent's user in to the WCM system at the right point of time (e.g. in a run level higher than or equal to `SERVER_UP`).

```
public boolean start(String arg0, int arg1, int arg2, String arg3)
{

      EventDispatcher eventDispatchM = VipRuntime.getEventDispatcher();
      eventDispatchM.addListener( Event.RUNLEVEL_IS, this );
      eventDispatchM.addListener( Event.OBJECT_METADATA_CHANGED, this );

      return true;
}
```

In the `performVipEvent` method, the respective events are processed.

As soon as the `SERVER_UP` run level is reached, the configured user is logged in by means of the `dologin` method. As the login is trivial, it is not explained here.

The `doSetup` method reads the assignments between users and languages saved in the `translation_lang_user` parameter. Via the method `AdminHandler.getUserProfile`, the user profile is loaded for each user ID and saved together with the locale in a map (`editorsLanguagesM`). From the user profile, the e-mail addresses of the translators will be retrieved later on.

However, if the event indicates a change to the metadata of an object, the `createTranslationObjects` method is called. This method ensures that a new object is created for each language supported. Afterwards, the editor responsible for translation is informed by e-mail.

```
public void performVipEvent(Event event)
{
      int eventId = event.getId();
      switch (eventId )
      {
         case Event.RUNLEVEL_IS :
            Integer runlevel =
                           (Integer)event.getArgument(Event.ARG_NEW);
            if ( runlevel.intValue() >= Runlevel.SERVER_UP &&
                                                        (!bLoginM ))
            {
               doLogin();
```

```
                doSetup();
            }
            break;

         case Event.OBJECT_METADATA_CHANGED :
            createTranslationObjects( event );
            break;

         default:
            break;
      }
}

....

private void doSetup()
{
      StringTokenizer loc_user_token = new StringTokenizer(loc_usersM,
                                                            ",", false);
      editorsLanguagesM = new HashMap();
      languagesM = new ArrayList();
      try
      {
         AdminHandler ah = VipRuntime.getAdminHandler();
         while ( loc_user_token.hasMoreElements() )
         {
            String lang = loc_user_token.nextToken();
            LocaleValue loc = new LocaleValue(lang);
            String userName = loc_user_token.nextToken();
            User translator = ah.getUserProfile( loginCidM, userName,
                                 AdminHandler.DIRECT_ASSIGNMENTS);
            editorsLanguagesM.put( loc, translator);
            languagesM.add(loc);
         }
      }
      catch ( VipApiException vaex )
      {
         System.out.println( "unable to start TranslationJobAgent..");
         vaex.printStackTrace();
      }
}
```

From the data of the event, the `createTranslationObjects` method first determines the OID and the website of the object for which the `OBJECT_METADATA_CHANGED` event was fired. Via the `ObjectHandler`, an `ObjectData` object is retrieved for the respective OID to check - on the basis of the `translate` attribute - whether the object is to be translated.

If the `translate` attribute has the value `true`, the content of the object to be translated is loaded and the file is saved in the directory **\temp** of the server on which the agent is running. After this, the agent creates a new object for each target language below the root object of the website. The content of the new objects is the content of the source object. Afterwards, the `informEditor` method is called and e-mails are sent to the editors responsible for the translation. In the `informEditor` method, the respective e-mails are created and sent via the Mail API.

```
private void createTranslationObjects( Event event )
{
      try
      {
         ObjectId oidNew = (ObjectId)event.getArgument( Event.ARG_OID );
         String ws = (String)event.getConstraint();
         ObjectHandler oh = VipRuntime.getObjectHandler(ws);
         Website website = VipRuntime.getAdminHandler().getWebsite
                  ( loginCidM, ws, AdminHandler.INDIRECT_ASSIGNMENTS);
         ObjectData od = oh.get(loginCidM, oidNew);
         BooleanValue translate = ( BooleanValue) od.getAttribute
                                  (new StringValue("translate"), null);
         if ( translate!= null && ( translate.getBool() == true ) )
         {
            Server server = VipRuntime.getCurrentServer();
            String tempdir =
                     website.getTemporaryDirectory().getAbsolutePath();
            String contentFileName = "contentTranslation";
            contentFileName += Long.toString
                                        (System.currentTimeMillis() );
            String contentPathAndFile =
               convertPath( tempdir+File.separator+contentFileName );
            File tmpContent =  new File( contentPathAndFile );
            FileOutputStream content = new FileOutputStream(tmpContent);
            od.getContent(content);
            Iterator languageIter =
                              editorsLanguagesM.keySet().iterator();
```

```
            while ( languageIter.hasNext())
            {
               LocaleValue lang = ( LocaleValue ) languageIter.next();
               String objTitle =  od.getTitle() + "_"+ lang.toString();
               ObjectData object2Translate = oh.createInitialObjectData(
               new ObjectId("1"), od.getTemplate(), od.getObjectType(),
                                                          objTitle );
               object2Translate.setLocale( lang.getLocale() );
               ObjectId oidCreated = oh.create(loginCidM,
                         object2Translate, tmpContent, "translate !! ");
               informEditor( lang , oidCreated,
                         object2Translate.getObjectType(), tmpContent);
               }
            }
         }
         catch ( VipApiException vaex )
.....
}
...
```

Via the `MailHandler`, the `informEditor` method creates and sends an e-mail to the responsible editors. The e-mail addresses are retrieved via the user profile of the editor that was read when the agent was started. The e-mail contains the OID of the object to be translated that was created in the method `createTranslationObjects` and the information on the language of the translation. The content to be translated is added to the e-mail as attachment.

```
private void informEditor( LocaleValue loc, ObjectId oid, ObjectType ot,
File content)
      throws VipApiException
{
      System.out.println( loc );
      MailHandler mh = VipRuntime.getMailHandler();
      EMailData emd = mh.createEMailData();
      User translator = ( User ) editorsLanguagesM.get( loc );
      String emailRec = translator.getEMailAddress();
      emd.setSubject("translation job for object with oid "+
                                                   oid.toString() );
      emd.setBody("the attached content must be translated into "+
                  loc.getLocale().getDisplayLanguage());
      emd.addRecipient(emailRec);
      emd.addAttachment( content, "content for Object "+ oid.toString(),
                                                   ot.getMimeType());
```

```
      try
      {
         mh.sendEMail(loginCidM, emd);
      }
      catch ( VipApiException vaex )
      {
         System.out.println(" could not send email to editor " +
                                                              emailRec );
         vaex.printStackTrace();
      }
}
```

Glossary

Access control list – For each WCM object, users, groups, roles, and group-roles that have access to this object can be specified. The individual access rights are specified separately for each principal authorized to access the object. Also called ACL.

ACL – Access Control List

Action – Step required to manage website objects. It generally results in a change in status of the object in question.

Activity – Workflow step to which a principal (user, group, or role) is assigned. An activity step represents the tasks of the assigned principal. There are three types of activities: “Edit”, “QA”, and “Delete”.

Agent – Additional function module based on the WCM Java API of Livelink WCM Server. Agents extend the standard functionality of the WCM servers.

Asynchronous action – An asynchronous action returns almost immediately after it has been called. It does not wait for processing of the associated tasks to finish. See also *Synchronous action*.

Check in – Staging action of Livelink WCM Server. An object that has been checked out and edited is returned to the WCM system by the action “Check in”. This makes the changes visible in the Edit view. The object is combined with the template again and is available for further editing.

Check out – Staging action of Livelink WCM Server. The content of a WCM object can only be edited after the object has been checked out. A checked-out object is locked for access by other users. The changes to the object do not become available in the Edit view until the object has been checked in.

Context – Contexts are used to identify users accessing the system or transactions. They are managed via the `ContextHandler` interface. A distinction is made between user and transaction contexts.

Context ID – Object that is assigned to a user after successfully logging in to the WCM system. A context ID is always unique throughout the entire system. It thus precisely identifies a user. If a context ID is not used over a certain period of time, it expires.

Data storage view – The data storage view of a server refers to the aspects of the WCM objects that are currently available (Edit view, QA view, Production view). The data storage view available on a server is determined in the routing settings of the website.

Deployment – Deployment is the distribution of data. The deployment of Livelink WCM Server performs two main tasks: first, generating pages from the WCM objects stored in the database and distributing the generated files to the appropriate directories; second, notifying the WCM servers of changes in the WCM system.

DeploymentEvent – An event that indicates activities (on searching and receiving WCM systems) relating to the distribution of data between WCM servers.

DeploymentHandler – Instance that actively initiates deployment actions.

Deployment system – The deployment systems generate pages from the WCM objects and distribute the generated files to the appropriate directories. From there, the files become visible for the users via an HTTP server. Deployment systems may be of various types and categories.

Deployment system category – Depending on the way of processing deployment jobs, deployment systems are assigned to various categories: *Standard deployment systems* automatically generate a new page every time a WCM object is changed. The generated pages are stored in the file system. *Dynamic deployment systems* generate the pages on the basis of user-defined settings and only when the page is requested via the HTTP server. The generated files are stored in a flat file structure. By means of

Search engine deployment systems, you can prepare your website data for use with a search engine. *WebDAV deployment systems* are required for the use of WebDAV clients. *InSite Editing deployment systems* provide the basis for editing and adding content directly in the website – without the Content client.

Deployment system types – On the basis of the staging concept of Livelink WCM Server, a distinction is made between deployment systems of type “Edit”, “QA”, and “Production”. Different views of the website data are generated, depending on the type.

Edit view – In the Edit view of Livelink WCM Server, the objects of a website are created and edited. Here the most current status of the objects is visible.

End point – When a WCM object is forwarded in the workflow and an end point is reached, the assignment between WCM object and workflow is removed.

Event – Events are fired by the WCM system when the state of a WCM object or a generated page or the system state changes.

Navigation topology – Structure of WCM objects within a website, organized according to topics (topic structure)

ObjectHandler – This instance accepts actions and thus manipulates the objects of the website.

Object category – Assignment of a WCM object to a specific category. Due to this assignment, the WCM object has a set of additional special attributes (metadata).

Object data – Components of a WCM object: content, metadata, and log

Object type – The specific kind of object, e.g. “HTML page”, “HTML template”, “Topic”. Various properties of the WCM object result from the object type. The object type is defined when the object is created. There are only a few cases in which it may subsequently be changed. Object types can be edited in the Admin client or the Content client.

Prepare event – A prepare event is fired before an action is performed. This makes it possible to check the action and, if necessary, prevent it by a `VetoException`.

Production view – The Production view of Livelink WCM Server makes the released pages of a website available to the user. By means of a web server, these pages can be accessed in the Internet, intranet, or extranet.

QA view – The QA view of Livelink WCM Server is used for quality assurance of the objects and thus of the website content. This view thus performs the control function between editing in the Edit view and publication in the Production view.

Release – Staging action of Livelink WCM Server. Quality Assurance checks whether the content and the form of a submitted object meet the quality standards of the company. If so, the object is released. The release transmits the quality-assured version of the object to the Production view, and thus makes it available to the end user in the published website.

Server agent – A Java implementation based on the WCM Java API. Server agents can expand or supplement the functionality of the WCM servers.

Server event – Occurrence that indicates an internal processing step in the WCM server in connection with object management

Session – Unit managed by the JSP engine so that logically related actions (in terms of resources) can be combined

Starting point – Initial point of a workflow. When a workflow is assigned to a WCM object, the object is automatically forwarded to the first activity.

Status – The processing state of a WCM object. Changes to the state are caused by the corresponding actions on the WCM object.

Submit – Staging action of Livelink WCM Server. before a newly created or changed object can be published, it must be submitted to Quality Assurance for checking. This makes the changes to the object visible in the QA view.

Synchronous action – A synchronous action does not return until all the tasks involved are completed. See also *Asynchronous action*.

Template topology – Structure of WCM objects within a website, organized according to templates (template structure)

Topology – A hierarchic organization of WCM objects according to specific criteria. In addition to the *Navigation topology* (organization according to topics with their respective subordinate objects) there is also a *Template topology*.

Transaction – A transaction can be used to group together several actions so that none of the changes involved actually becomes effective until the transaction is finalized (committed).

Transition – Connection between two activities or a starting/end point and an activity in a workflow. By means of the transitions, the WCM objects are forwarded within the workflow.

VIPP – VIP Protocol. A proprietary protocol for exchanging data between the components of a WCM system. VIPP can be tunneled in HTTP for communication in WANs or over the Internet.

Workflow definition – Description of a workflow. A workflow definition consists of starting points, end points, activities, and transitions. Workflow definitions are saved in XPDL format and can be edited graphically by means of the Content Workflow Modeler.

Index

A

access control
    ACL 141
access control lists
    edit 141
access rights
    for a Livelink object 189
    for a WCM object 107
    use (of parent topic) 123
access to Livelink WCM Server
    via remote API 30
    via server agents 22
ACL (interface) 141
ACL_OWNER (search attribute) 138
actions
    and deployment 118
    execute in transactions 113
    of object management 109
Activity (interface) 143
add
    functional area 72
    Livelink object 194
    object 109
    principal 54
    relator 199
addRemark (method) 109
AdminHandler (interface) 51, 157
administration rights
    constants of VipAdminPermission 65
    edit (of principals) 65
    example for editing 68
agents 22
    constructor 24
    events 30
    ServerAgent interface 24
    start 27

stop 29
analyzing deployment orders 168
application examples 205
assignments
of principals 74
retrieve 75
substitute 75
attribute sets 128
AttributeHandler (interface) 128
attributes 128
attribute sets 128
object categories 129
authentication 37

C

change
run level 173
change (method) 109
check in 109
check out 109
undo 109
checkIn (method) 109
checkout (method) 109
classes
BooleanValue 132
DateValue 132
DoubleValue 132
ExternalObjectRelator 197
IntegerValue 132
Link 132
ListValue 132
LivelinkObjectFactory 194
LivelinkRelator 197
LivelinkRuntime 182
LocaleValue 132
LongValue 132
message classes 42
ObjectHandlerUtil 120
ObjectId 124, 132

PoolConnection 45
PrincipalFilter 137
properties files 42
RepositoryEntry 123
RootTemplateFilter 137
SetValue 132
StringValue 132
SubtreeFilter 137
Text 132
TimeValue 132
UserMessage 42
Version 124
VipApiException 40
VipRuntime 20
class-specific tracing 177
constants
for administration rights 65
for data storage views 100
for functional areas 72
for object rights 69
constructor for server agents 24
content 126
add to Livelink object 194
edit of relators 202
edit of WCM objects 126
read of Livelink objects 192
Content Workflow Modeler 103
ContentConverter (interface) 146
context management 36
contexts 117
predefined 38
refresh 39
convert objects 109, 146
convertContent (method) 109
converter
Executable 146
copy 109
copy (method) 109
create (method) 109

D

data storage views
    constants 100
default object rights
    edit (of principals) 69
    example 70
delete
    functional area 72
    Livelink object 195
    page 110
    principal 61
    relator 203
    WCM object 110
delete (method) 110
deployment
    and actions 118
    DeploymentHandler interface 165
    errors 171
    events 87
    job execution 169
    metadata 166
    order analysis 168
    status information 164, 167
    wait for completion 119
deployment systems
    information 164
DEPLOYMENT_SYSTEM_CREATED (event) 88
DEPLOYMENT_SYSTEM_DELETED (event) 88
DeploymentError (interface) 172
DeploymentEvent (interface) 87
DeploymentEventListener (interface) 91
DeploymentHandler (interface) 165
DeploymentMetaData (interface) 166
DeploymentPrepareEvent (interface) 90
DeploymentPrepareEventListener (interface) 92
DeploymentStatus (interface) 167
DeploymentSystem (interface) 164
DeploymentWaitInfo 119
depublishPage (method) 110

destroy (method) 110
direct release 109
directRelease (method) 109
documentation
    Javadoc 12

E

edit
    access control list 141
    assignments 74
    attributes 128
    content of relators 202
    content of WCM objects 126
    deployment-specific metadata of relators 201
    functional areas 71
    Livelink metadata of relators 200
    object data 121
    principals 54
    profiles of principals 57
    references 125
    relator 200
    rights of principals 64
    standard metadata 122
    WCM objects 95
edit objects
    actions of the ObjectHandler 109
e-mail
    adapt layout 151
    compile 154
    configuration file 151
    generate via WCM Java API 154
    HTML format 154
    plain text 154
    send 155
    specify content 151
    status change 150
e-mail support 149
EmailData (interface) 149
Entity (interface) 143
error messages 40

localization 41
errors
deployment 171
Event (interface) 83
event processing 81
event dispatchers 92
event listeners 90
EventDispatcher (interface) 92
EventListener (interface) 91
events 83
deployment-related 87
preparation events 89
system-related 85
website-related 84
examples 205
automatic submission 209
convert special characters 221
monitoring deployment events 215
vetoing metadata changes 212
exceptions 40
translated messages 41
VipApiException basic class 40
execute deployment jobs 169
external object 196
ExternalObject (interface) 196
ExternalObjectRelator (class) 197

F

factory
FunctionalArea 72
PrincipalFactory 54
file
instantiate 147
file type
File 147
FILE_CHANGED (events) 88
FILE_DELETED (event) 88
FILE_URL_CHANGED (event) 88
filter 133

attribute filter 134
Filter (interface) 133
filter (method) 109
functional areas 71
add and delete 72
assign to principal 74
constants 72
information 73
FunctionalArea (interface) 72

G

generatePage (method) 110
generateTopic (method) 110
get (method) 109
getPrincipals (method) 58
global tracing 177
Group (interface) 56
groups 54
add 54
delete 61
edit assignments 74
edit rights 64
Group interface 56
profile 56
search 58

I

import 109
information
on deployment 165
on deployment errors 171
on deployment systems 164
on functional areas 73
on pages 166
on servers 160
on the WCM system 157
on websites 162
interface
Acl 141

Activity 143
AdminHandler 51
AttributeHandler 128
ContentConverter 146
ContextHandler 36
DeploymentError 172
DeploymentEvent 87
DeploymentEventListener 91
DeploymentHandler 165
DeploymentMetaData 166
DeploymentPrepareEvent 90
DeploymentPrepareEventListener 92
DeploymentStatus 167
DeploymentSystem 164
EmailData 149
Entity 143
Event 83
EventDispatcher 92
EventListener 91
ExternalObject 196
Filter 133
FunctionalArea 72
Group 56
Key 131
Link 125
LivelinkAdminHandler 183
LivelinkObjectHandler 184
LivelinkObjectType 190
LivelinkVersion 193
MailHandler 149
ObjectData 121
ObjectHandler 108, 109
ObjectLoader 127
ObjectState 97
ObjectType 105
PoolManager 44
PrepareEvent 89
PrepareEventListener 91
Principal 53
PrincipalFactory 54
Progress 115
RemoteRuntime 31

Role 56
Runlevel 173
Runtime 31
SearchableKeys 137
Server 160
SystemHandler 157
TraceFilter 176
Transition 144
User 56
Value 131
VipAdminPermission 65
VipObjectPermission 69, 141
Website 162
Workflow 142
WorkflowHandler 143

J

Java API
access via remote API 30
access via server agents 22
Javadoc 12
job execution (deployment) 169

K

Key (interface) 131

L

LDAP
set WCM attributes 61
links 125
Livelink
navigating 188
object management 184
object types 189
user administration 183
Livelink integration 181
Livelink object
add 194
add content 194
cache 187

change connection 199
change metadata 195
check permissions 189
delete 195
establish connection 198
identify 185
log 192
read 186
read content 192
read log entries 192
reserve 195
retrieve metadata 191
unreserve 195
LivelinkAdminHandler (interface) 183
LivelinkObjectFactory (class) 194
LivelinkObjectHandler (interface) 184
LivelinkObjectType (Interface) 190
LivelinkRelator (class) 197
LivelinkRuntime (class) 182
LivelinkVersion (interface) 193
load
several objects 127
localization
error messages 41
messages 42
logged-in users 160
logging in 37
logging out 37
login 37
logout 37
logs
of Livelink objects 192
of servers 174
of WCM objects 125
read server logs 174

M

MailHandler (interface) 149
message files 42

messages
localization 42
translation 42
metadata 122, 128
edit of relators 200
edit of WCM objects 109
of a Livelink object 191
of deployment 166
methods
synchronous and asynchronous 111
MIME type 154
move 109
move (method) 109
multiImport (method) 109

N

new
functional area 72
Livelink object 194
object 109
principal 54
relator 199

O

object
add 109
add remark 109
check in 109
check out 109
convert 109, 146
copy 109
create (method) 109
delete 110
edit 95
edit metadata 109
events after actions 84
find (filter) 133
get for editing 109
identify (Livelink) 185
import 109

move 109
reject 109
release 109
restore old version 109
submit 109
undo check out 109
object category
edit 129
object data 121
attributes 128
content 126
log 125
OID 124
references 125
standard metadata 122
version 124
object management 95
object rights 107
constants of VipObjectPermission 69
object status 97
object types 105
OBJECT_ACL_CHANGED (event) 84
OBJECT_CHECKED_IN (event) 84
OBJECT_CHECKED_OUT (event) 84
OBJECT_CHECKOUT_UNDONE (event) 84
OBJECT_CONTENT_SYNCHRONIZED (event) 85
OBJECT_CREATED (event) 84
OBJECT_DELETED (event) 84
OBJECT_DESTROYED (event) 84
OBJECT_EXPIRED (event) 84
OBJECT_METADATA_CHANGED (event) 84
OBJECT_REJECTED_TO_EDIT (event) 84
OBJECT_SUBMIT_AT_TO_PRODUCTION (event) 85
OBJECT_SUBMITTED_TO_PRODUCTION (event) 85
OBJECT_SUBMITTED_TO_QA (event) 85
OBJECT_TRANSACTION_END (event) 85
OBJECT_WORKFLOW_ASSIGNED (event) 85
OBJECT_WORKFLOW_FORWARDED (event) 85

OBJECT_WORKFLOW_REMOVED (event) 85
ObjectData (interface) 121
ObjectHandler (class) 120
ObjectHandler (interface) 108
ObjectId (class) 124
ObjectLoader (interface) 127
ObjectState (interface) 97
ObjectType (interface) 105
old version
    restore 109
order analysis (deployment) 168

P

page
    delete 110
    information 166
    regenerate 110
    remove 110
path of the page 166
pool management 44
PoolConnection 45
PooManager (interface) 44
preparation events 89
PREPARE_ASSIGN_WORKFLOW (event) 89
PREPARE_CHECKIN_OBJECT (event) 89
PREPARE_CHECKOUT_OBJECT (event) 89
PREPARE_CREATE_OBJECT (event) 89
PREPARE_DELETE_OBJECT (event) 89
PREPARE_DESTROY_OBJECT (event) 89
PREPARE_FILE_CHANGED (event) 90
PREPARE_FORWARD_WORKFLOW (event) 89
PREPARE_OBJECT_CHANGE_METADATA (event) 89
PREPARE_REJECT_OBJECT_TO_EDIT (event) 89
PREPARE_RELEASE_OBJECT (event) 89
PREPARE_REMOVE_WORKFLOW (event) 89
PREPARE_SUBMIT_OBJECT_TO_QA (event) 89
PREPARE_UNDO_CHECKOUT_OBJECT (events) 89

PrepareEvent (interface) 89
PrepareEventListener (interface) 91
Principal (interface) 53
PrincipalFactory (interface) 54
principals 54
    add 54
    administration rights 65
    delete 61
    edit assignments 74
    edit profiles 56, 57
    edit rights 64
    retrieve profiles 56
    search 58
    WCM attributes for LDAP principals 61
profiles
    edit 57
    of principals 56
    retrieve 56
Progress (interface) 115
properties
    of deployment systems 164
    of functional areas 73
    of servers 160
    of users 56
    of websites 162
properties files 42

R

read OID 124
references 125
reject 109
reject (method) 109
relator 196
    add 199
    delete 203
    edit 200
    edit content 202
    edit deployment-specific metadata 201
    edit Livelink metadata 200

release 109
release (method) 109
remark
  add 109
remote API 30
  request 32
  request via SSL 33
RemoteRuntime (interface) 31
remove
  functional area 72
  principal 61
reports 175
reports of servers 175
RepositoryEntry 123
reserve (Livelink object) 195
restoreVersion (method) 109
retrieve
  assignments 75
  functional areas 73
  profiles of principals 56
rights 107
  edit 64
  for administrators of the WCM system 65
Role (interface) 56
roles 54
  add 54
  delete 61
  edit assignments 74
  edit rights 64
  profile 56
  Role interface 56
  search 58
run level
  set 173
Runlevel (interface) 173
RUNLEVEL_DECREASES (event) 85
RUNLEVEL_INCREASES (event) 85
RUNLEVEL_IS (event) 85
Runtime (interface) 31

S

search
- additional search attributes 137
- predefined search function 137
- principals 58
- search parameters 139
- WCM objects 133

search attributes
- ACL_OWNER 138
- STATE_NAME 138
- TYPE_NAME 138

SearchableKeys (interface) 137

Server (interface) 160

server administration 160

server agents 22
- configuration 23
- constructor 24
- events 30
- ServerAgent interface 24
- start 27
- stop 29

servers
- information 160
- read logs 174
- read reports 175
- set run level 173
- tracing 176

several objects
- load 127

special attributes 128

staging views
- constants 100

standard metadata 122

start (server agents) 27

STATE_NAME (search attribute) 138

status change
- configure e-mail 151
- generate e-mail 150

status information
    on deployment 167
stop (server agents) 29
submit 109
submit (method) 109
substitute
    assign 75
system administration 157
system information 157
SystemHandler (interface) 157
system-related events 85

T

TraceFilter (interface) 176
tracing 176
transaction context 117
transactions 113
    deployment 167
    progress control 115
Transition (interface) 144
translation
    error messages 41
    messages 42
tree operations 109
TYPE_NAME (search attribute) 138

U

undoCheckOut (method) 109
unreserve (Livelink object) 195
URL of the page 166
User (interface) 56
user administration 51
user authentication 37
user context 117
USER_LOGIN (event) 85
USER_LOGOUT (event) 85
UserMessage (class) 42

users 54
add 54
delete 61
edit assignments 74
edit rights 64
profile 56
retrieve logged-in 160
search 58
User interface 56

V

Value (interface) 131
version
read (of a WCM object) 124
views on data storage
constants 100
VipAdminPermission
example 68
VipAdminPermission (interface) 65
VipApiException 40
VipObjectPermission
constants 69
example 70
VipObjectPermission (interface) 69, 141

W

WCM object
edit 95
events after actions 84
find (filter) 133
WCM system
events 85
Website (interface) 162
WEBSITE_DELETED (event) 86
WEBSITE_NEW (event) 86
websites
assign to principal 74
information 162
set run level 173

workflow
activity 143
assign 144
forward object 145
remove assignment 146
transition 144
Workflow (interface) 142
workflow definition 103
WorkflowHandler (interface) 143